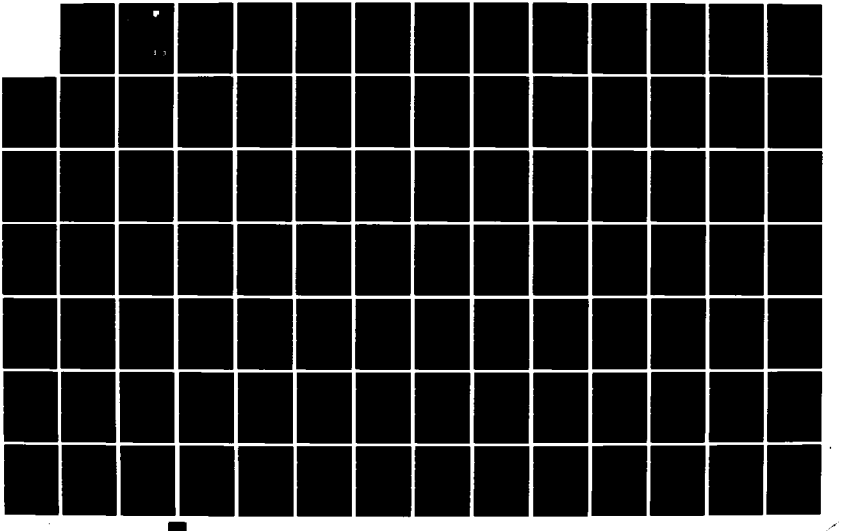
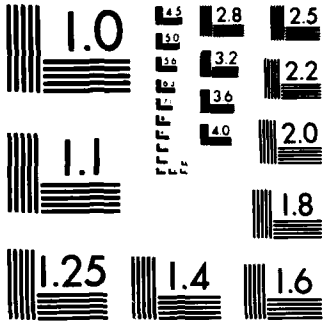
AD-A123 636 SOFTWARE RETEST TECHNIQUES(U) COMPUTER SCIENCES CORP- 1/2
FALLS CHURCH VA K FISCHER ET AL. OCT 82 RADC-TR-82-275
F30602-81-C-0089

UNCLASSIFIED F/G 9/2 NL






MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

RADC-TR-82-275
Final Technical Report
October 1982


# SOFTWARE RETEST TECHNIQUES

Computer Sciences Corporation

Dr. Kurt Fischer, Farzad Raji and Daniela Onaszko

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED*



**ROME AIR DEVELOPMENT CENTER**
**Air Force Systems Command**
**Griffiss Air Force Base, NY 13441**

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-82-275 has been reviewed and is approved for publication.

APPROVED: Andrew J Chruscicki

ANDREW J. CHRUSCICKI
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command & Control Division

FOR THE COMMANDER: John P Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>RADC-TR-82-275 | 2. GOVT ACCESSION NO.<br>AD-A113636 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>SOFTWARE RETEST TECHNIQUES | | 5. TYPE OF REPORT & PERIOD COVERED<br>Final Technical Report<br>Feb 81 - Feb 82 |
| | | 6. PERFORMING ORG. REPORT NUMBER<br>N/A |
| 7. AUTHOR(s)<br>Dr. Kurt Fischer<br>Farzad Raji<br>Daniela Onaszko | | 8. CONTRACT OR GRANT NUMBER(s)<br>F30602-81-C-0089 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Computer Sciences Corporation<br>6265 Arlington Blvd<br>Falls Church VA 22046 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>62702F<br>55811828 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Rome Air Development Center (COEE)<br>Griffiss AFB NY 13441 | | 12. REPORT DATE<br>October 1982 |
| | | 13. NUMBER OF PAGES<br>144 |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*<br>Same | | 15. SECURITY CLASS. *(of this report)*<br>UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE<br>N/A |

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribuiton unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Andrew J. Chruscicki (COEE)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Software Maintenance
Software Retest
Software Life Cycle Management
0-1 Integer Programming
Graph Theory

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The purpose of this effort was to study and then develop techniques for maintaining software systems. The report focuses on current maintenance problems, various strategies for retesting, besides an analysis of these strategies. A methodology for retesting was developed that generates the minimum number of test cases to validate a code modification. To generate the minimum number of test cases the methodology analyzes the program data and logic structure dependencies. The selected test cases assure re-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

testing of decision to decision paths (dd-paths) reaching the changed code and setting the changed data, and reached from the changed code and using the changed data.

TABLE OF CONTENTS


| Paragraph | | Page |
|---|---|---|
| Section 1 - Summary | | 1-1 |
| Section 2 - Introduction | | 2-1 |
| 2.1 | Background | 2-1 |
| 2.2 | Current State-of-the-Art of Software Maintenance | 2-6 |
| 2.2.1 | Management Issues | 2-8 |
| 2.2.2 | Graph Theory Issues | 2-12 |
| 2.2.3 | Technical Issues | 2-16 |
| 2.3 | Current Air Force Retest Practices | 2-19 |
| Section 3 - Overview of Retest Strategies | | 3-1 |
| 3.1 | Introduction | 3-1 |
| 3.1.1 | Retest Strategy 1: Rerun All Test Cases | 3-1 |
| 3.1.2 | Retest Strategy 2: Retest All Testable Paths Through the Changed Code | 3-2 |
| 3.1.3 | Retest Strategy 3: Rerun All Test Cases Which Execute the Changed Code | 3-3 |
| 3.1.4 | Retest Strategy 4: Retest All DD-Paths Reached From the Changed Code | 3-3 |
| 3.1.5 | Retest Strategy 5: Retest All DD-paths Reaching To and Reached From the the Changed Code | 3-8 |
| 3.1.6 | Retest Strategy 6: Retest All DD-paths Reaching the Changed Code and Setting Changed Data, and Reached From the Changed Code and Using Changed Data | 3-8 |
| 3.2 | Detailed Description of Retest Strategies | 3-12 |
| 3.2.1 | Strategy 1 | 3-14 |
| 3.2.2 | Strategy 2 | 3-14 |
| 3.2.3 | Strategy 3 | 3-14 |
| 3.2.4 | Strategy 4 | 3-16 |
| 3.2.5 | Strategy 5 | 3-23 |
| 3.2.6 | Strategy 6 | 3-27 |
| 3.2.6.1 | Methods and Procedures | 3-27 |
| 3.2.6.2 | Manual Walkthrough of Set/Use Matrix | 3-33 |
| 3.2.6.2.1 | Manual Walkthrough of Algorithm A | 3-34 |
| 3.2.6.2.2 | Manual Walkthrough of Algorithm B | 3-35 |
| 3.2.6.3 | Live Example of Strategy 6 | 3-36 |
| 3.2.7 | System Level Analysis | 3-46 |
| 3.2.7.1 | Global Set/Use Table | 3-46 |
| 3.2.7.2 | Usage of Example Global Set/Use Table | 3-49 |

Paragraph Page

3.3 Result Obtained..... 3-50
3.3.1 Strategy Prioritization..... 3-50
3.3.2 Methodology Characteristics..... 3-54

Section 4 - Conclusion..... 4-1

Section 5 - Recommendations..... 5-1

Appendix A - Data Dependency Analysis Algorithm A..... A-1

Appendix B - Data Dependency Analysis Algorithm B..... B-1

Appendix C - Bibliography..... C-1

Appendix D - Technical Paper..... D-1

Appendix E - Glossary..... E-1

Appendix F - Glossary of Terms..... F-1

Appendix G - Glossary of Acronyms..... G-1

## FIGURES


| Number | | Page |
|---|---|---|
| 2-1 | Organizational Responsibility and Data Flow for Retest Process | 2-22 |
| 3-1 | Path Enumeration Illustration | 3-2 |
| 3-2 | Simple Code Modification and Test Cases | 3-4 |
| 3-3 | Flow Diagram with Test Cases | 3-5 |
| 3-4 | Sample Flowchart | 3-7 |
| 3-5 | Sample Flowchart | 3-10 |
| 3-6 | Sample Flowchart (Strategy 6) | 3-11 |
| 3-7 | Sample Source Code | 3-12 |
| 3-8 | Sample Application Flowchart and Test Paths | 3-15 |
| 3-9 | Sample Retest Formulation | 3-20 |
| 3-10 | Sample Retest Model Formulation After Application of Reduction Rule 1 | 3-22 |
| 3-11 | Sample Retest Model Formulation After Application of Reduction Rule 2 | 3-22 |
| 3-12 | Example Formulation | 3-25 |
| 3-13 | Comparison of the Results of the Application of Strategy 4 vs. Sratetgy 5 | 3-28 |
| 3-14 | Sample Set/Use Matrix | 3-29 |
| 3-15 | Example Formulation | 3-31 |
| 3-16 | Logical AND Operation (Step 1) | 3-30 |
| 3-17 | Logical AND Operation (Step II) | 3-33 |
| 3-18 | Logical OR Operation | 3-33 |
| 3-19 | Example Set/Use Matrix | 3-34 |
| 3-20 | Set/Use Matrix After Applying Algorithm A | 3-34 |
| 3-21 | Set/Use Matrix After Applying Algorithm B | 3-35 |
| 3-22 | Graph Presentation of Data Statement Module and Identification of DD-paths | 3-37 |
| 3-23 | Logical AND Operation for Data Statement Module | 3-36 |
| 3-24 | Logical AND Operation for Dat. Statement Module | 3-43 |
| 3-25 | Logical OR Operation for Data Statement Module | 3-43 |
| 3-26 | 0-1 Integer Programming Model for Data Statement | 3-44 |
| 3-27 | Reduced 0-1 Integer Programming Model For Data Statement Module | 3-45 |
| 3-28 | Probability of Feasible Testcase Selection Without Application of the Retest Methodology For Data Statement Module | 3-48 |

## FIGURES - APPENDIX D


| Number | | Page |
|---|---|---|
| 1 | Directed Graph Presentation of a Module..... | D-6 |
| 2 | Connectivity Matrix..... | D-7 |
| 3 | Reachability Matrix..... | D-8 |
| 4 | Test Case Cross Reference Matrix..... | D-8 |
| 5 | Module Set/Use Matrix..... | D-9 |
| 6 | 0-1 Integer Programming Model..... | D-9 |
| 7 | Logical AND Operation..... | D-11 |
| 8 | Logical AND Process..... | D-12 |
| 9 | Logical OR Operation..... | D-12 |
| 10 | Global Variable Set/Use Matrix..... | D-13 |


## APPENDIX F


| | | |
|---|---|---|
| 1-1 | Diagram of a Branch(1)..... | F-2 |

# TABLES


| Number | | Page |
|---|---|---|
| 3-1 | Retest Strategies | 3-13 |
| 3-2 | Sample Test Case Cross Reference Matrix | 3-14 |
| 3-3 | Example Test Case Cross Reference Matrix | 3-21 |
| 3-4 | Example Connectivity Matrix | 3-21 |
| 3-5 | Example Reachability Matrix | 3-21 |
| 3-6 | Example Test Case Cross Reference Matrix | 3-26 |
| 3-7 | Example Reachability Matrix | 3-26 |
| 3-8 | Example Test Case Cross Reference Matrix | 3-32 |
| 3-9 | Example Reachability Matrix | 3-32 |
| 3-10 | Sample Set/Use Matrix | 3-32 |
| 3-11 | Reachability Matrix for Data Statement Module | 3-38 |
| 3-12 | Test Case Cross Reference Matrix For Data Statement Module | 3-39 |
| 3-13 | Set/Use Matrix For Data Statement Module | 3-40 |
| 3-14 | Set/Use Matrix For Data Statement Module After Applying Algorithm A | 3-41 |
| 3-15 | Set/Use Matrix For Data Statement Module After Applying Algorithm B | 3-42 |
| 3-16 | Alternative Optimum Combinations of Test Cases For Data Statement Module Modification | 3-47 |
| 3-17 | Global Variable Set/Use Matrix | 3-49 |
| 3-18 | Retest Strategy Priority | 3-52 |

# SECTION 1 - SUMMARY

Computer Sciences Corporation (CSC), under contract with Rome Air Development Center (RADC), developed a methodology to retest modified software during the operation and support phase of the software acquisition life cycle. The Statement of Work tasks and a brief summary of our accomplishments for each task follow:

Task 1: Investigate existing techniques and methodologies applicable to the retesting of software during the operation and support phase of software development. The investigation shall include, but not be limited to, applicable technology which is currently being utilized in various Air Force Operational Environments. Observed deficiencies with current technology shall be identified and described. Software retesting activities to be addressed shall include those involved with the identification of computer program components (routines, paths, variables, etc.) affected by a specified modification, and the subsequent application of necessary testing technology to verify the integrity of the modified software.

State-of-the-art techniques and software tools which are purported to provide automated aids for program development, debugging, testing, retesting, maintenance and documentation shall be examined and evaluated to determine applicability of the techniques and the extent to which they provide retesting support. Applicable software tools residing in industry, university, and Government environments shall be examined. Included shall be an examination of the facilities provided by the JOVIAL J73 Automated Verification System (J73 AVS) (Reference the Functional Description (CR-1-947) dated March 1980).

Summary of Work Performed Under Task 1: Different methodologies and techniques such as path analysis , decision

to decision path (DD-path) analysis, (see Appendix F for definition), graph theoretic approach, and software segmentation were investigated and the feasibility of each in terms of applicability to Air Force software applications, reliability, and cost-effectiveness was studied.

A software retest methodology was developed which considers the impact of modifications to both the control structure as well as the data dependency of a computer program. The retest methodology selects the minimum number of test cases needed to validate software modifications by applying DD-path analysis, graph theory, and an optimization technique.

The facilities for static and dynamic analysis (e.g., path analyses, set/use analysis) provided by the JOVIAL J73 Automated Verification System were studied and incorporated into the functional design of the automated Software Retest System. Additionally, various optimization techniques were evaluated and the 0-1 integer programming package currently available as part of the system support software at major computer centers was selected and incorporated into the automated Software Retest System.

Task 2: Develop and describe advanced techniques and methodologies which can be automated to enhance existing Air Force software retesting capability. The retesting technology described shall be applicable to the Air Force approved Higher Order Languages.

The techniques and methodologies described shall be prioritized according to:

1. Effectiveness in supporting retesting requirements.
2. Degree of automation possible.
3. Ease of implementation.
4. Cost of implementation.
5. Cost of application (computer resources, manpower, etc.).
6. Reliability.

Summary of Work Performed Under Task 2: Under this task, alternative retest strategies were defined and models built for their automated implementation. The defined strategies are:

1. Rerun all previously executed tests.
2. Retest all testable paths through the changed code.
3. Rerun all tests which execute the changed code.
4. Retest all DD-paths reachable from the changed code.
5. Retest all DD-paths reaching to and reachable from the changed code.
6. Retest all DD-paths reaching to the changed code and setting changed data, and reached from the changed code and using changed data.

Though strategy 2 was shown to be impractical, techniques were developed to implement each remaining strategy. Implementation techniques for strategies 1 and 3 were based on test execution history, while implementation techniques for strategies 4 and 5 were developed using more sophisticated techniques. For these strategies, the logical structure of the source code was transformed into a directed graph and the graph analyzed in terms of each strategy. For strategy 6, both the logical and the data structure of the code were used to build the retest model. The optimization technique of 0-1 integer programming was then applied to minimize the amount of retesting within the constraints inherent in each strategy. The retest methodology and Software Retest System were developed to be compatible with the existing Air Force software packages (e.g., Jovial J73 Automated Verification System) and High Order Languages. The Software Retest System could be implemented in any of the Air Force High Order Languages without losing language independence in the methodology.

We have also studied the characteristics of the alternative strategies that are relevant to the Air Force needs. These

characteristics are prioritized and presented in Section 3 of this report.

Task 3: Techniques (described in response to Task 2, Statement of Work Paragraph 4.1.2) which are considered cost effective and mature enough to be implemented in future software tools and successfully utilized in support of Air Force Operational Software shall be formally specified. A functional description, in accordance with the CDRL, shall be prepared.

Summary of Work Performed Under Task 3: An automated version of the Software Retest Methodology incorporating Strategy 6 was formally specified and a functional description was prepared.

Software Retest Project Deliverable Items

In addition to this technical report, a functional description for the automated Software Retest System (SRS) has been defined. Also, a technical paper was presented at the National Telecommunication Conference '81. A copy of this paper is contained in Appendix D.

# SECTION 2 - INTRODUCTION

## 2.1 BACKGROUND

Computer software is the non-hardware portion of a computer or information system, and includes computer code (instructions), and documentation. The development of computer software usually goes through an evolutionary life cycle beginning with the establishment of a user need, and ending with the use or operation of the computer program or information system. The individual phases of the Air Force Software Acquisition Life Cycle follow the steps shown below:

1. Conceptual Phase - Software Requirements Analysis
2. Validation Phase - Specification Development and Study
3. Full scale Development Phase - Preliminary and Detailed Design
4. Production Phase - Coding and Testing
5. Deployment Phase - Operations and Maintenance

The 1970's brought much progress in the fields of software management and software engineering to many phases of the software life cycle. The early part of the decade brought significant increases in programmer productivity with such techniques as structured programming, top-down development, code walkthroughs, chief programmer teams, HIPO charts, automated testing aids, and automated documentation aids. These tools and techniques, though quite useful, only concentrated on the detailed design, code, and test phases of the software life cycle. In addition, both software customers and vendors recognized the importance of a firm, complete, and accurate understanding of the software requirements. Automated tools to perform completeness and consistency checking such as the University of Michigan's ISDOS and Ballistic Missile Defense Advanced Technology Center's Software Requirements

Engineering Methodology (SREM) are becoming more widely used in both business and scientific types of applications.

The Deployment Phase (Operations and Maintenance) begins with delivery of the first operational unit and terminates when the system is removed from the operational inventory. Changes to computer programs are made to remove latent errors, improve coding or operation, adapt to changes in system requirements, or incorporate knowledge gained from operational use.

Studying the individual life cycle phases and their dependencies can be very important as the U.S. economy is currently spending about $20 billion annually in the area of computer software (2). A small increase in productivity can be extremely cost effective. In the past, many software projects have taken short cuts during the early life cycle phases so that a product could be quickly fielded. Boehm (2) has reported that the cost of correcting an error during the design phase is only half the cost of correcting an error during the coding phase, and only a tenth the cost of correcting an error found during the acceptance test phase. During the 1960's, however, software developers paid little attention to life cycle modeling as software customers seemed to be more concerned about quick delivery than error-free code.

Recent research in software management has shown the importance of the Deployment Phase of the software life cycle.

---

1. Teichroew, D., "ISDOS and Recent Extensions," Proceedings of the Symposium on Computer Software Engineering, Polytechnic Press (1976), p. 79.

2. Boehm, B.W., "Software Engineering," IEEE Trans. on Computers, Vol. C-25, No. 12, December 1976, pp. 1226-1242.

Several authors have reported that maintenance costs account for between 40-60% of the system life cycle cost (1,2). One survey of predominantly business data processing managers found that 90% of the respondents ranked maintenance of equal or greater importance than new system design (3). Unfortunately, the increased importance in software maintenance has not been accompanied by new or improved methods for performing and managing the maintenance task.

One technical problem area, called retest, arises when attempting to validate code modifications. Retest is the act of testing existing software after modification. It differs from the test activity, which is concerned with planning and executing tests that initially validate the entire software system. Retest deals with the following problems:

1. How can it be shown that a change to one area of the code does not create data and/or logic conditions that could affect the proper execution of another area?

2. Do the previously used test cases need to be rerun?

   If so, how many, and what subset?

3. Do the modifications require generation of new test cases? If yes, how many?

The problem of what to retest and how thoroughly to do so is a major problem for software managers and researchers and has not yet been adequately resolved either through research or

---

1. Boehm, B.W., op. cit.

2. Canning, R.G., "That Maintenance Iceberg," EDP Analyzer, Vol. 10, No. 10 (1972), pp. 1-14.

3. Lindhorst, M.W., "Scheduled Maintenance of Applications Software," Datamation, Vol. 19, No. 5, (1973), pp. 64-67.

through accepted management practice. In the area of research, no work has been published identifying a retest methodology. In the area of management practice, retest decisions still appear to be made ad hoc. The purist will demand that all previously executed tests be rerun. The pragmatist will leave the decision to the discretion on the test team's technical leader (often called the test director) as he believes the test director knows the software best, and by using engineering judgment and his knowledge of the code he often manually selects the subset of previously completed tests to be rerun. Other plausible retest methods may be: to rerun a number of randomly selected tests, to rerun all tests that execute the modified code, or to execute a new set of test cases (sometimes called confidence test cases) that exercise all the program's major capabilities to give the user "confidence" (though not statistically) that the software operates properly.

Each method has some beneficial properties, yet none gives a perfectly reliable solution. Rerunning all previously used test cases is almost always impractical as the validation process for large computer programs may take several man-years. The test director may be able to select for retest those tests that address the functional modifications, but he may not be aware that modified data conditions could cause execution of non-functional paths resulting in inaccurate output that may go undetected for years. What is needed is a quantitative method for assuring that new program modifications are correct and do not introduce new errors into the code. To formally prove this would take an analysis of every program path, but this has been shown to be an impractical task in all but the most trivial cases (1).

---

1. Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5 (1973), pp. 48-59.

The research explored during this project addressed the validation of software modifications. The general question was, "How can software modifications be validated?", and more specifically:

1. What quantitative techniques can be used to implement these strategies?
2. Are these techniques feasible?
3. What are the implementation considerations?

These questions are critical to software practitioners, because the ad hoc retest selection techniques of the past have proved woefully inadequate.

Choosing a subset of test cases to be rerun can be done using either quantitative or ad hoc techniques. No other known studies have comprehensively addressed either method, though in practice, the latter is most often employed. The approach taken during this project was to apply graph theory to the analysis of software modifications. This approach has been successful in the area of initial software testing and was extended here to analyze the retest problem.

The goal of this research was to find methods to perform retesting in the most efficient and reliable way. With this methodology, we hoped to be able to answer questions such as:

1. "What parts of the software system need to be retested after modification?"
2. "How much retesting is needed?"
3. "What are the most efficient methods of retesting?"
4. "What test cases need to be rerun?"

The first step in the research was to define alternative retest strategies which assure specific retest coverage. Conclusions were drawn based on the performance of each strategy and generalized beyond the research environment.

Developments in the retest area will advance computer software theory as well as benefit the management of the software maintenance process. The current graph theoretic approach to computer program testing will be extended to computer program maintenance. Practical benefits are anticipated to be significant because now maintainers will have a tool with which to make tradeoff analyses with regard to cost versus test coverage. In addition, managers will be able to have increased confidence that modifications to one module do not affect the proper execution of the entire software system.

## 2.2 CURRENT STATE-OF-THE-ART OF SOFTWARE MAINTENANCE

The maintenance effort for many software systems today runs from 40-60 percent of the system life cycle cost (1,2). Canning's studies of B. F. Goodrich and General Motors (2) show the need for an increase of maintenance in the data processing environment. Canning's report estimates that up to 80 percent of the effort at Oldsmobile is maintenance. The Boehm (3) report on two Air Force Command and Control projects indicates the maintenance portion of the 10 years life cycle cost is about 67-72 percent. In a recent report to Congress, it was estimated that the Government spends 1.3 billion dollars on software maintenance. Not included in this estimate is

---

1. Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," op. cit.

2. Canning, R.G., op. cit.

3. Boehm B.W., "Software Engineering," op. cit.

the software maintenance cost associated with embedded weapons systems (1).

The need for software modification during the operations and maintenance phase of the software life cycle is unavoidable and the retesting of these modifications is essential. Unfortunately, there are few tools available to assist software maintenance personnel in determining the proper retesting procedures. This is not for lack of need, however. Boehm (2) and Lipow (3) discuss the uncertain reliability of software subsequent to maintenance modifications. Gibson and Railing (4), Donahoo and Swearingen (5), Yau and Collofello (6),

---

1. Report to the Congress of the United States, "Federal Agencies' Maintenance of Computer Program: Expensive and Undermanaged", February of 1981.

2. Boehm B.W., "Software Engineering," op. cit.

3. Lipow, M., "Some Directed Graph Methods for Analyzing Computer Program," Proceedings, Computer Sciences and Statistics: Eighth Annual Symposium on the Interface, Health Sciences Computing Facility, UCLA, February 1975.

4. Gibson, C.G. and L.R. Railing, "Verification Guidelines," TRW Software Series #71-04, August 1971.

5. Donahoo, J. D. and D. Swearingen, A Review of Software Maintenance Technology, RADC-TR-80-13, Rome Air Development Center, Griffiss AFB, NY, February 1980.

6. Yau, S.S. and J. Collofello, "Some Stability Measures for Software Maintenance," IEEE Transaction Software Engineering, Vol. SE-6, No. 6, November 1980.

and Liu (1) specifically identify the need for developing a formal quantitative maintenance validation procedure.

Three areas of software maintenance and retesting were reviewed and significant findings in each area are discussed in this section. The management procedures for software maintenance and the need for software retesting tools and technology is discussed in 2.2.1. Utilization of graph theory as a solution to retesting problems is discussed in 2.2.2, and difficulties involved with software maintenance and retesting are identified in 2.2.3.

These areas provided sufficient information for the development of our retest methodology.

### 2.2.1 Management Issues

For better employee morale and more efficient maintenance, Lindhorst (2) suggests a "scheduled maintenance" approach for maintenance of application software. Scheduled maintenance is a policy whereby maintenance is deferred until a predetermined month when all maintenance modifications for an application are performed. The article attributes the following benefits to the utilization of this approach:

1. Consolidation of requests.
2. Programmer job enrichment.
3. Better user analysis prior to the request for modification.
4. Periodic application program evaluation.
5. Elimination of "squeaky wheel syndrome".

---

1. Liu, C.C., "A Look at Software Maintenance," Datamation, Vol. 22, No. 11, (1976), pp. 51-55.

2. Lindhorst, M.W., op. cit.

6. Programmer back-up.

7. Better planning.

Mooney (1) suggests "organized program maintenance" to produce more efficient and reliable software modifications. The organized program maintenance approach suggests that motivational factors such as increased salaries and rotation to a software development team after 6 months will significantly increase the productivity and morale of a maintenance team.

Swanson (2) categorizes the failure of software into three types: process failure, performance failure, and implementation failure. Based on this categorization, he suggests an "organizational structure" approach to software maintenance. This approach uses a team of programmers whose only responsibility is the maintenance of installed software. The team uses a maintenance data base and every change must be made through a maintenance order. Swanson believes that utilization of these tools, will result in better software maintenance management and less software failure.

Boehm (3) categorizes software modifications during maintenance into: software updates which results in a change of specification, and software repair which does not affect the software specification.

---

1. Mooney, J.W., "Organized Program Maintenance," Datamation, Vol. 21, No. 2 (1975), pp. 63-64.

2. Swanson, E.B., "The Dimensions of Maintenance," Proceedings, Second International Conference on Software Engineering, IEEE Catalog 76CH1125-4C, October 1976, pp. 492-497.

3. Boehm, B.W., "Software Engineering," op. cit.

Managing software maintenance has problems similar to managing any other activity. Ignoring the problems leads to poor software maintenance and software process failure thereby increasing the cost of maintenance. Boehm (1) enumerates software management problems as follows:

1. Poor planning.
2. Poor control.
3. Poor resource estimation.
4. Unsuitable management personnel.
5. Poor accountability structure.
6. Inappropriate success criteria.
7. Procrastination on key activities.

To overcome these problems, he states that the manager of a software maintenance activity must keep the maintenance team current with state-of-the-art technology, especially in the area of software tools. Such tools could be an automated software retest system similar to the one described in the "Software Retest System Functional Description" that can increase the reliability of modified or repaired software and decrease failures.

Yau and Collofello (2) discuss software maintenance for large-scale software. They break down the software maintenance modification process into different phases and steps. Their defined maintenance process is a set of phases; each phase and its associated process is critical to the maintenance process. Before beginning a phase and making any modifications, their maintenance process requires that maintenance objectives be

---

1. Boehm, B.W., "Software Engineering," op. cit.

2. Yau, S.S. and J. Collofello, "Some Stability Measures for Software Maintenance," IEEE Transaction Software Engineering, Vol. SE-6, No. 6, November 1980.

determined so that maintenance personnel understand what to modify. The "maintenance process" phases are:

Phase 1- Understanding the program with regard to the program's complexity and self descriptiveness. The complexity of a program is a measure of the effort required to understand the program. Self descriptiveness of the program is a measure of the clarity of the program.

Phase 2- Generating a particular maintenance proposal, keeping in mind extensibility, which is a measure of the extent to which a program can support extensions or critical functions.

Phase 3- Accounting for ripple effect since the affect of a modification may not only be local to the modification, but may also affect other portions of the program.

Phase 4- Testing to assure the modified program has at least the same reliability as it had prior to modification. Once Phase 4 is completed, Yau and Collofello recommend determining the success of retesting effort. If it is determined to be unsuccessful, the maintenance modification objectives must be reevaluated and the maintenance process repeated.

One of the most critical and neglected aspects of software maintenance and development is the human factor. This phase of the software life cycle needs experts that can quickly understand existing software and that can rapidly modify software. Unfortunately, software maintenance is not viewed as the most exciting portion of the software life cycle and frequently software maintenance personnel experience boredom with their jobs. Mooney (1) experienced this problem and his solution was to rotate personnel between a development team and a maintenance team. Additionally, a pay increase for

---

1. Mooney, J.W., op. cit.

maintenance team members was provided. This method significantly increased job satisfaction.

Shneiderman (1) considers the working environment as the major factor influencing the behavior of programmers and maintainers. The physical environmental factors found to be the most significant are:

1. Room size.
2. Room structure (window, door, ceiling, etc.).
3. Brightness of the light.
4. Air temperature and humidity.
5. Arrangement of desk and work space.
6. Access to computer terminal and facilities.
7. Noise quality and intensity.
8. Interference from others.
9. Degree of privacy.

He concluded that a poor working environment decreases the quality and quantity of the programmer/designer's work.

### 2.2.2 Graph Theory Issues

Graph theory has played a heavy role in two very specialized areas of software testing: logic path generation, and program structure analysis. The identification of logic paths is used by those who have built automatic test data generator programs. These programs have discovered some well hidden errors. Hoffman (2) discusses his experience with the

---

1. Schneiderman, B., op. cit.

2. Hoffman, R. H., "The Impossible Pairs Detection Capability (IMPAIR) of the Automated Test Data Generator (ATDG)," NASA, Contract No. NAS9-14853, Houston, Texas, January 14, 1977.

Impossible Transfer Pairs Detection Capability, (IMPAIR). This system was tested with his Automatic Test Data Generator program and as a result of running 15 test cases, two function errors were found and four system modifications had to be made. Other researchers, such as Fosdick and Osterweil (1), have developed static analysis programs which analyze logic paths to assure data consistency and software reliability. Shooman and Ruston (2) propose an "analytical determination of program paths." They present an algorithm based on labeling branches of a program with a binary number that identifies the number of possible paths in a program. As a result, each path is a combination of branches with a unique binary number.

Graph theory is also utilized in the area of program structure analysis. By using graph theory, it is possible to determine the degree to which a program complies with various coding constructs. Brown and Fischer (3) introduce a technique called "segmentation" which involves analysis of program source code. Based on an algorithm, a tool was developed to audit source code for compliance with structured programming

---

1. Fosdick, L.D. and L.J. Osterweil, "DAVE - A Fortran Program Analysis System," Proceedings, Computer Science and Statistics: Eighth Annual Symposium on the Interface, Health Sciences Computing Facility, UCLA, February 1975, pp. 329-335.

2. Shooman, M.L. and H. Ruston, "Summary of Technical Progress Investigation of Software Models," Rome Air Development Center, RADC-TR-79-188, Griffiss AFB, NY.

3. Brown, J.R. and K.F. Fischer, "A Graph Theoretic Approach to the Verification of Program Structures," Proceedings, Third International Conference on Software Engineering, IEEE Catalog No. 78CH1317-7C, May 1978.

constructs. Gannon and Else (1) discuss the utilization of program branches (DD-paths) in computer program analysis.

There is other research that discuss graph theoretic approaches to program testing. Krause, Smith, and Goodwin (2) give an introduction on the use of graph theory in testing, and discuss a method of designing test cases to exercise all of the code using source code analysis, base path and loop generation, optimal path design, and user interface.

Huang (3) gives an excellent tutorial on program testing from a graph theoretic viewpoint. Lipow (4) discusses a graph theoretic approach to testing by using Dilworth's theorem on partially ordered sets to determine the minimum number of testcases needed to execute all segments of a computer program at least once. Miller (5) identifies three major categories for program testing technology: theoretical functions, methodology, and automated tools. Additionally, he identifies over twenty program testing "needs" from a graph theory

---

1. Gannon, C. and R. F. Else, "JOVIAL J73 Automated Verification System User's Manual," General Research Corporation, July 1981.

2. Krause, K.W., R.W. Smith and M.A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis," Record, 1973, IEEE Symposium on Computer Software Reliability, New York, 1973, pp. 18-22.

3. Huang, J.C., "An Approach to Program Testing," Computing Surveys, Vol. 7, No. 3 (1975), pp. 113-128.

4. Lipow, M., op. cit.

5. Miller, R.E., "Program Testing Technology in 1980's," Proceedings of the Conference on Computing in the 1980's, IEEE, 1978.

viewpoint and describes each one in detail. Gannon (1) conducted an experiment in which she compared two software testing techniques: static analysis and dynamic path testing. Both tools ran in a similar environment. The result indicates that between the two testing techniques, dynamic path testing is the most effective at detecting logic, computational, and data base errors. In this experiment, dynamic path testing detected 25 percent of the seeded errors.

Voges, Gmeiner and Amschler (2) designed and implemented an automated testing tool capable of testing a single FORTRAN module. This tool views the module as a directed graph and generates test cases which require at least one execution of each DD-path. Ntafos and Hakimi (3) introduces algorithms for covering a minimum set of paths during program testing. Paige (4) views the computer program as a graph structure and discusses different approaches to partitioning program graphs.

---


1. Gannon, C., "Error Detection Using Path Testing and Statistic Analysis," IEEE Transactions on Computer, August, 1979.

2. Voges, U., Gmeiner, and Amscher, "SADAT, an Automated Testing Tool", IEEE Transaction on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 286-290.

3. Ntafos, S.C. and S.L. Hakimi, "On Path Problems in Diagraphs and Application to Program Testing; IEEE Transaction on Software Engineering, Vol. SE5, No. 5, September 1979.

4. Paige, M.R. "On Partitioning Program Graph", IEEE Transaction on Software Engineering, Vol SE-3, No. 6, November 1977.

Sloane (1) presents an algorithm for finding the paths through a network which could be applied to computer programs. Fischer (2) used a graph theoretic approach to determine the minimum number of previously executed test cases needed to retest every reachable program segment subsequent to code modification.

### 2.2.3 Technical Issues

As previously discussed, between 40 and 60 percent of the system life cycle is spent on maintenance. The major technical problems involved in computer program maintenance concern the lack of software tools for maintenance, the focus of software research, and the reliability of software modifications.

To effectively and efficiently perform the maintenance task, tools are needed. Unfortunately, however, most of the available tools were developed for use during software production, not maintenance. Moreover, our literature review and on-site surveys performed as part of this effort, indicate that few tools are procured by organizations responsible for software maintenance.

Another technical problem which affects software maintenance is the direction and focus of research. Boehm (3) has defined the following categories in which maintenance research should be targeted:

1. Understanding the existing software.

---

1. Sloan, N.J.A., "On Finding the Paths Through a Network," The Bell System Technical Journal, Vol. 51, No. 2 (1972), pp. 371-390.

2. Fischer, K. F., "A Test Case Selection Method for the Validation of Software Maintenance Modifications," Proceedings, COMPSAC '77, IEEE, November 1977, pp. 421-426.

3. Boehm, B.W., "Software Engineering," op. cit.

2. Modifying the existing software.

3. Reevaluating the modified software.

4. General aids.

The reliability of post-modified software has long been a technical issue among software maintainers. The prevailing assumption is that the reliability of the software is directly associated with the reliability of the testing. Tai (1) analyzes the complexity of programs and measures that complexity by the amount of test data required for demonstrating program correctness by testing. Based on this complexity measurement, he introduces new test selection criteria. To select test data, Howden (2) compares five methods of software testing with the following results:

| Method | Errors Found in the Same Program |
|---|---|
| Path | 18 |
| Branch | 6 |
| Structured | 12 |
| Special Value | 17 |
| Symbolic | 17 |

Fischer (3) proposed the use of "quality assurance software tools". This is a useful technique to increase the reliability

---

1. Tai, K., "Program Testing Complexity and Test Criteria," IEEE Transaction on Software Engineering, Vol. SE-6, No. 6, November 1980.

2. Howden, W.E., "Methodology for the Generation of Program Test Data," IEEE Transaction on Computers, Vol. C-24, No. 5 (1975), pp. 554-559.

3. Fischer, K. F., The FORTRAN Code Auditor, Quality Assurance Software Tools User's Guide, TRW Software Product Assurance, STP-6039, January 1977.

of the software, by using tools such as code auditors, path analyzers, variable analyzers, etc.

Changing software during maintenance raises the question, "Is the performance of the software changed?" The answer can be found in a ripple effect analysis study performed by Yau and Collofello (1). The first part of the study identifies program areas which require additional maintenance to insure consistency with the initial change. The second part of the study analyzes how changes to one program area affects the performance of other program areas. Ripple effect analysis is performed in three steps:

1. Change management system.

   In this step, maintenance personnel provide the system with source code, a proposed modification, and performance requirements. The system creates a record of changes in a data base.

2. Lexical analysis package.

   This step is performed once the modification to the program has been completed. The program is analyzed with respect to the proposed modification and a characterization of the program containing information necessary for tracing both logical and performance ripple effects is compiled and saved in a data base.

3. Tracing package.

   In this step, maintenance personnel execute the tracing package which utilizes the data base of program changes created by the change management system and maps these changes into the characterization of the program created by the previous step.

---

1. Yau, S.S. and J.S. Colofello, op. cit.

## 2.3 Current Air Force Retest Practices

As part of the "Software Retest Techniques" contract, on-site interviews were conducted to determine retesting practices currently being used by the Air Force. Four programs were selected for participation:

1. Short Range Attack Missile Program, Oklahoma Air Logistics Center.
2. 427M System, North American Air Defence Command, Space Computation Center.
3. F-111 Operational Flight Program, Sacramento Air Logistics Center.
4. Communication Software, Oklahoma Air Logistics Center.

Based on these interviews, 3 retest practices were identified:

1. Selection of test cases to validate modifications to software were made by the staff based on their knowledge and familiarity with the software.
2. Manual techniques were used to select testcases.
3. One large test case was used to validate all modifications to the software.

Each practice offers some beneficial properties, yet none provide a reliable, cost-effective solution.

In the first practice, the selection of test cases to validate a software modification is made subjectively by personnel based on their knowledge of the software and the test bed. Given that such personnel are available, they may be able to select those test cases that address functional modifications. However, without extensive knowledge of the software, they may be unable to select test cases that address

data conditions that can cause execution of non-functional paths. The execution of non-functional paths leads to inaccurate output which decreases system reliability. Furthermore, since retest decisions are made subjectively, statistical confidence that the software operates properly can not be given. A software program utilizing this retest approach will be only as reliable as the personnel responsible for making retest decisions.

In the second practice, the selection of test cases to validate software modifications is made without the benefits derived from the utilization of automated support tools. Therefore, the inputs (analysis of data and logic dependencies) used to select test cases are developed manually. Since the inputs are generated manually a greater probability of error is introduced. Additionally, the cost and time of manually generating inputs is greater than the automated generation of inputs.

In the third practice, one large test case is used to validate modifications made to the software. This large test case is developed to exercise the entire system baseline, not just the modified portion of the baseline. The major concern with this practice involves the manpower and computer costs associated with testing those portions of the baseline that need not be tested as thoroughly as the modified areas of the baseline. Additionally, since it is not feasible to retest each modification, a modification cycle is frequently used. For example, during an 18-month modification cycle, several modifications may need to be made to the software. If the need for a modification is identified during month 1 of the cycle, it will not be implemented or tested until month 18. The time lag associated with such a modification cycle may be unacceptable as well as frustrating to users.

The organizational responsibilities and the data flow for the retest process are shown in Figure 2-1. As indicated, a review board or configuration management office has approval authority for the modification and its validation (retest) while the software support group performs the actual modification and retest.

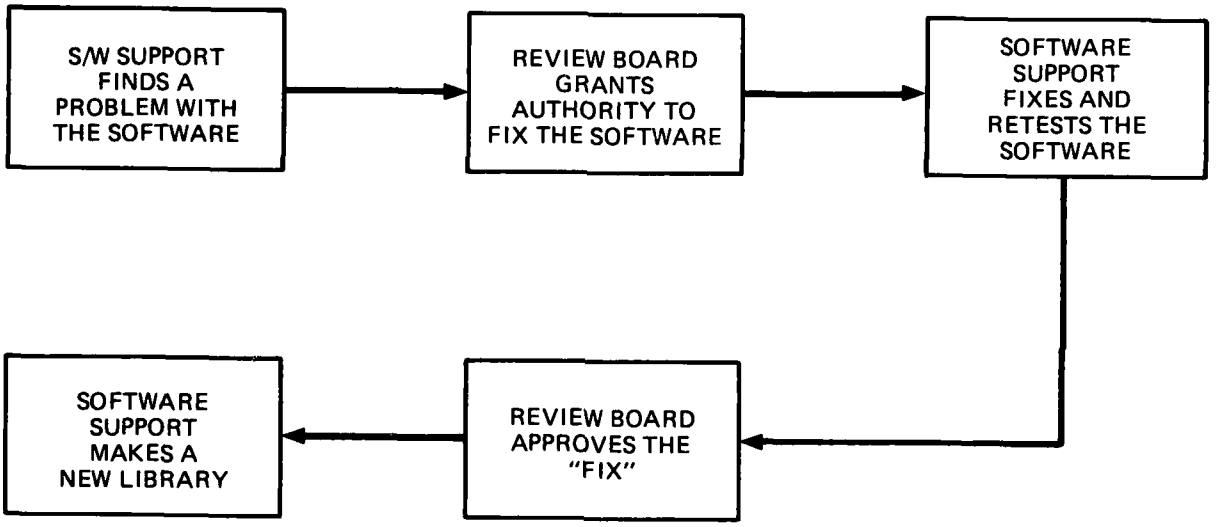



Figure 2-1. Organizational Responsibilities and Data Flow for the Retest Process

# SECTION 3 - OVERVIEW OF RETEST STRATEGIES

## 3.1 INTRODUCTION

Approaches to retesting, and the degree to which it is performed will vary widely depending on the goals and constraints on a given software maintenance environment. Managers usually have good intentions, but there are no widely accepted retest standards or methodologies. Retest strategies typically amount to a shotgun approach of performing as much retesting as possible within set schedule and cost constraints.

This section describes six alternative retest strategies which explicitly define the amount of retesting to be performed for any given code change. Examples using the selected strategies are given along with algorithms for their implementation.

### 3.1.1 Retest Strategy 1: Rerun All Test Cases

In terms of selection effort, the easiest retest strategy is to rerun all previously used test cases. In the extreme best-case, the set of previously executed test cases provides full test coverage of existing capability. If the test bed executes all DD-paths, then this strategy will usually provide an overkill of the effort required to validate small software modifications. While this strategy may be convenient for small programs where the number of test cases is low, it may not be feasible for medium to large systems where the number of test cases is high. Even if the number of test cases is low, rerunning all test cases will not guarantee software quality unless in the aggregate the test cases provide full test coverage. In a large software system where the number of test cases may exceed one thousand, it is too time consuming and expensive to rerun all test cases. Additionally, the set of previously executed test cases may not functionally or structurally test that code introduced as a result of the modification. Consequently, system reliability steadily decreases unbeknown to the software support group.

### 3.1.2 Retest Strategy 2: Retest All Testable Paths Through the Changed Code

This strategy implies that one has identified and developed test cases for each testable path. While it is possible to identify the set of all paths using a graph (1), the number of paths through software containing loops may be very high, making it impractical to develop test cases for each path. To illustrate the large number of paths in a relatively simple graph, consider the example illustrated in Figure 3-1. If all

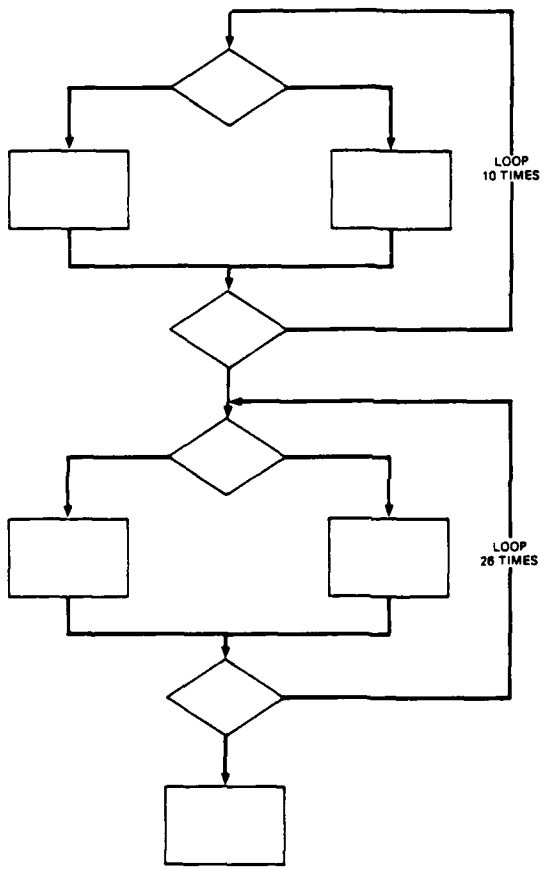



Figure 3-1. Path Enumeration Illustration.

1. Sloan, N.J.A., "On Finding the Paths Through a Network," The Bell System Technical Journal, Vol. 51, No. 2 (1972), pp. 371-390.

the branches are assumed to be independent there are 68,719,476,736 (calculated as $2^{10} \cdot 2^{26}$) paths in the graph. Even though all decision statements are seldom mutually independent, the number of testable paths is usually too high to consider building test cases for each. Therefore, this retest strategy was dropped from consideration because its implementation is currently infeasible.

### 3.1.3 Retest Strategy 3: Rerun All Test Cases Which Execute the Changed Code

Clearly, if a test case does not exercise the modified code, it is not impacted and need not be rerun. This strategy confines retesting to only those test cases that execute the modified code. For example, if there are four test cases to test the code represented in Figure 3-2, and DD-path 3 is modified, one can visually verify that test cases 1 and 2 need not be retested as they do not execute DD-path 3. In this example, only test cases 3 and 4 need be rerun. Implementing this strategy would reduce the required retesting by 50 percent for this particular example.

Though rerunning all tests which pass through the modified code provides high test coverage, it may also require more retesting than either the budget or schedule will allow. Consider the example depicted in Figure 3-3. Using the strategy of rerunning all test cases that execute the modification, a simple code modification to DD-path 3 would require rerunning 8 of the 12 possible test cases. For this example, two-thirds of the testbed would have to be rerun in order to validate a simple modification.

### 3.1.4 Retest Strategy 4: Retest All DD-Paths Reached From the Changed Code

Seven software entities (see Appendix F for definitions) identified with code are:

1. Programs
2. Modules

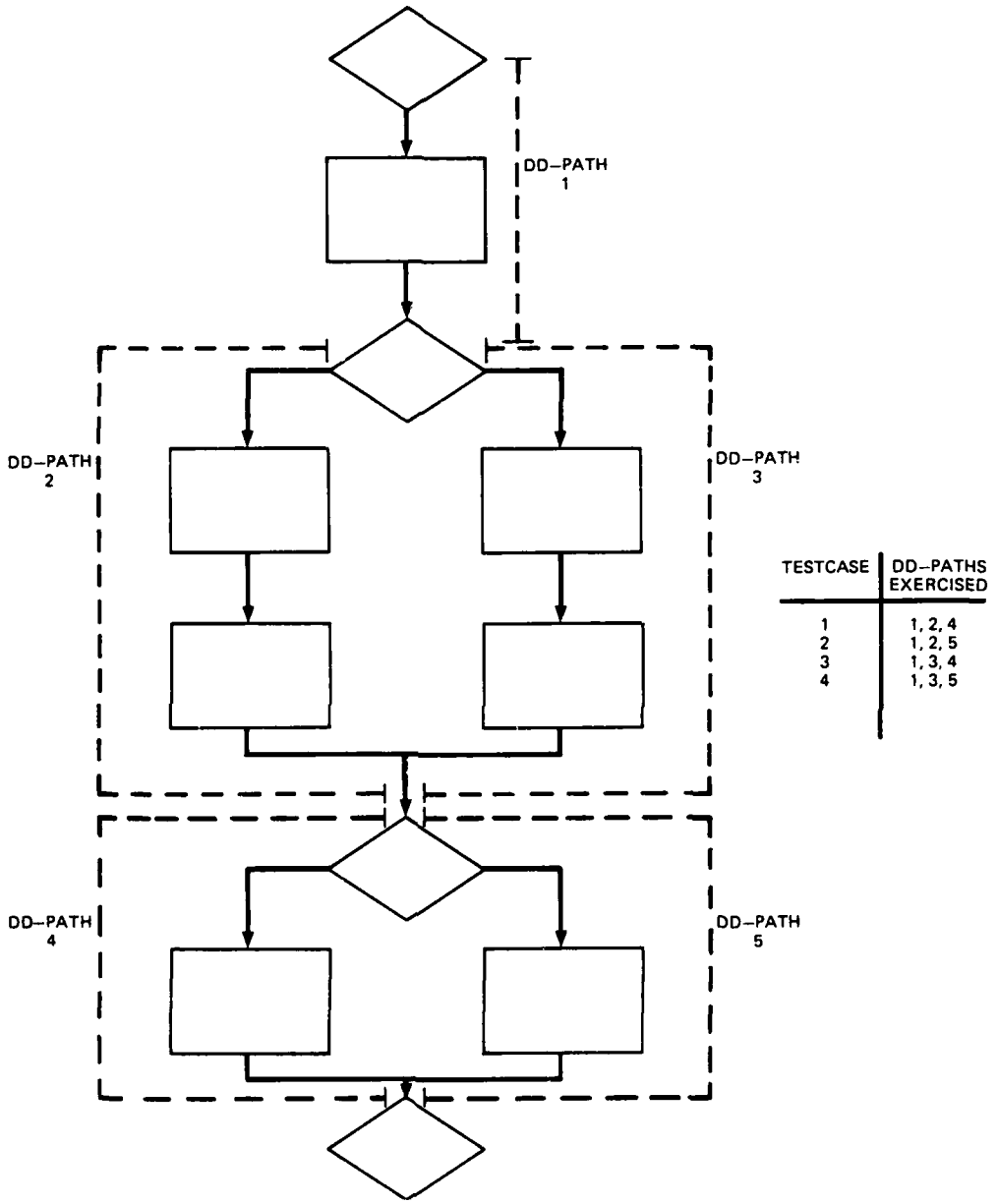
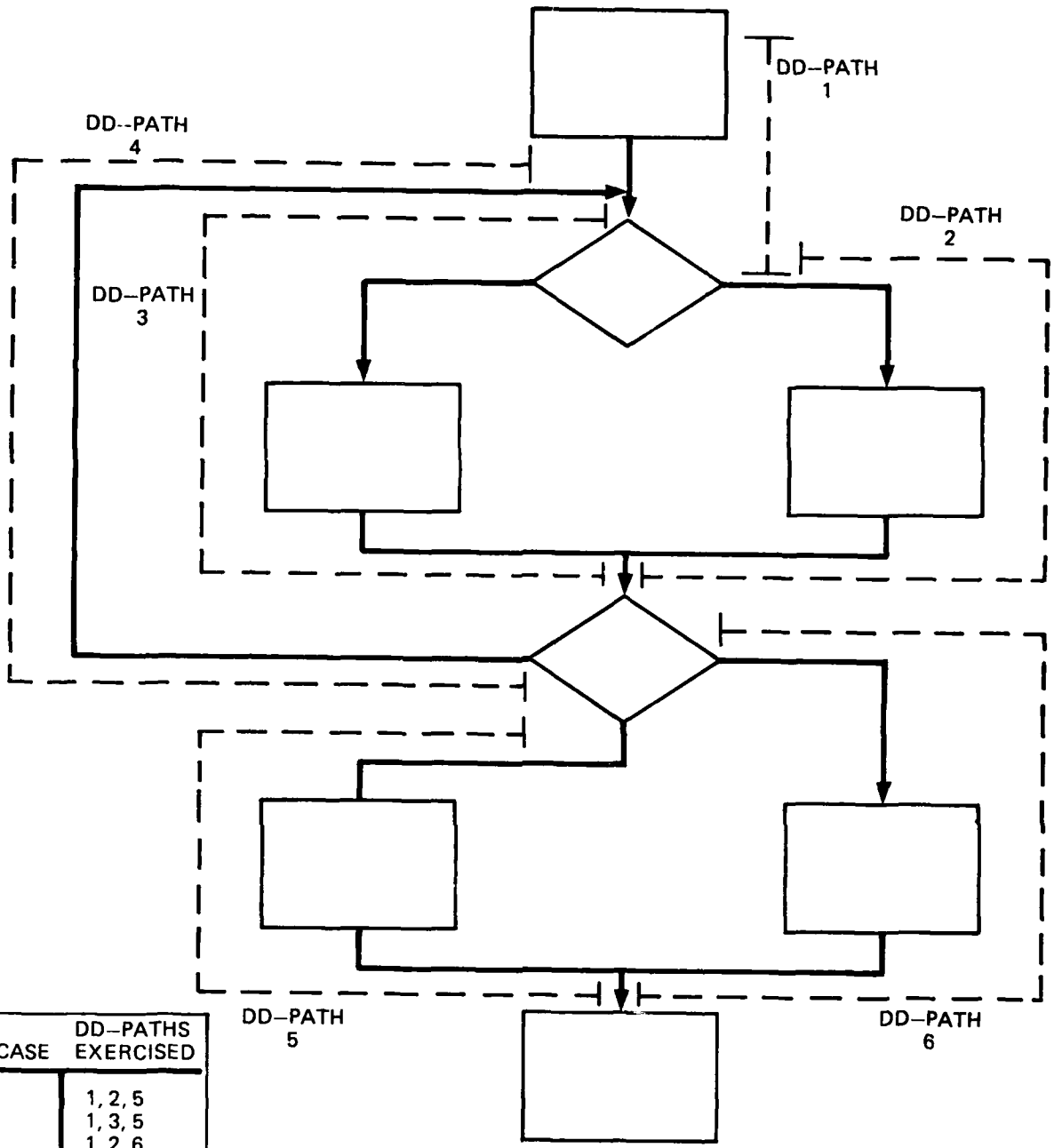



Figure 3-2. Simple Code Modification with Test Cases




| TEST CASE | DD-PATHS EXERCISED |
|---|---|
| 1 | 1, 2, 5 |
| 2 | 1, 3, 5 |
| 3 | 1, 2, 6 |
| 4 | 1, 3, 6 |
| 5 | 1, 2, 4, 2, 6 |
| 6 | 1, 2, 4, 2, 5 |
| 7 | 1, 2, 4, 3, 6 |
| 8 | 1, 2, 4, 3, 5 |
| 9 | 1, 3, 4, 2, 6 |
| 10 | 1, 3, 4, 2, 5 |
| 11 | 1, 3, 4, 3, 6 |
| 12 | 1, 3, 4, 3, 5 |

Figure 3-3. Flow Diagram With Test Cases

3. DD-paths

4. Segments

5. Statements

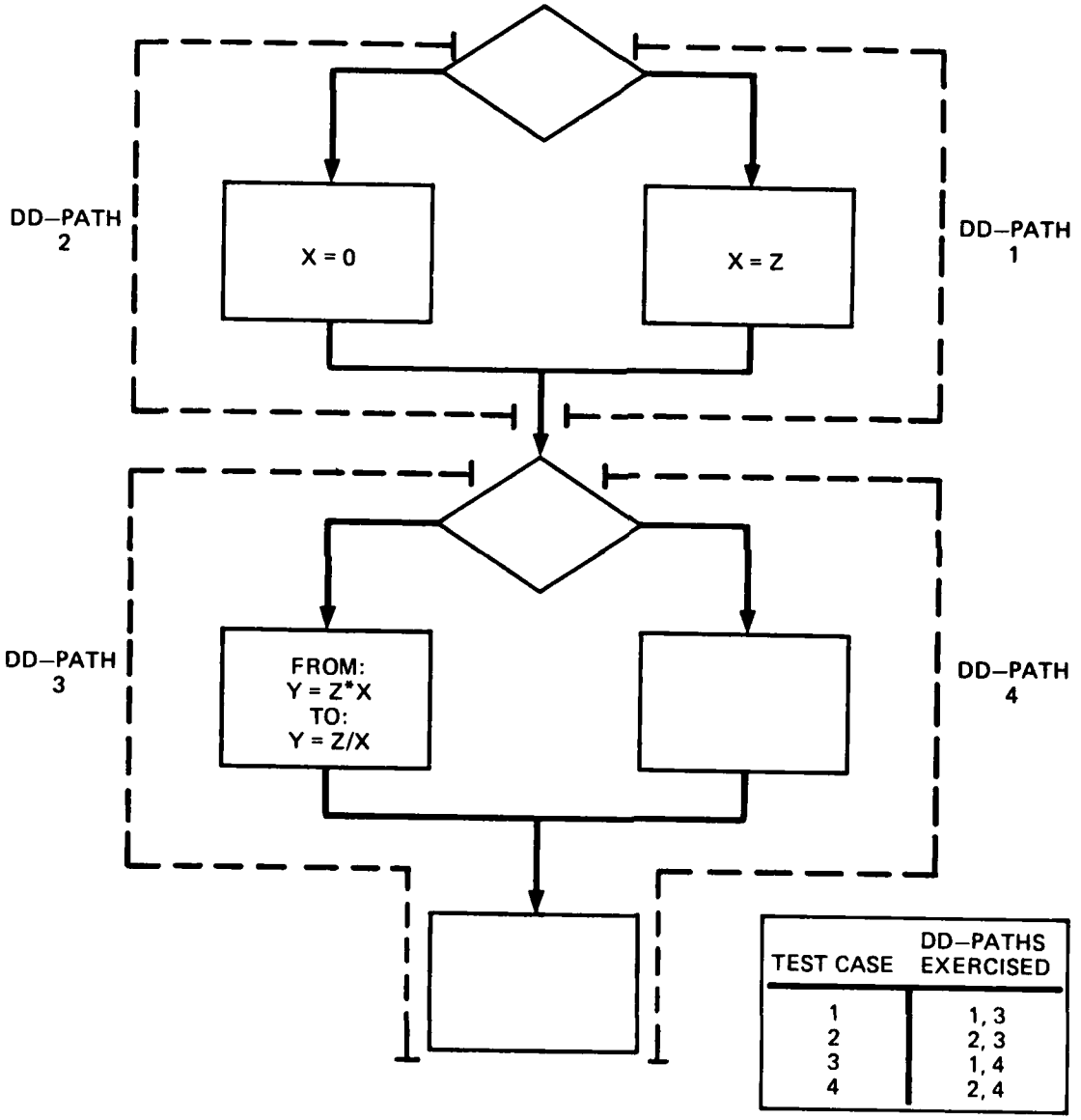
6. Branches

7. Paths

Merely entering and exiting the program (entity 1) does not provide sufficient retesting. On the other hand, sufficient resources to test every path (entity 7) may not be available and, as demonstrated in paragraph 3.1.2, is highly impractical if not an impossible task. We then search for a middle ground of software entities to sufficiently test. We know that we would like to test all branches and statements. The technique of breaking a computer program down into DD-paths allows for consideration of all branches and statements in a more manageable way. Therefore, it may be concluded that a reasonable retest strategy is to execute all DD-paths at least once. However, if this strategy is examined for efficiency, we find that it can be improved. For example, if a simple modification to DD-path 5 (in Figure 3-3) is made, it seems inefficient to retest DD-path 6 since the execution of DD-path 5 has no impact on DD-path 6. A better retest strategy would be to retest a subset of the testbed which covers all DD-paths that are reached from the modified code. This strategy incorporates the benefits of full test coverage, but eliminates retesting DD-paths that cannot be reached from the modified code. The minimum retest subset would then be any set of test cases such that all DD-paths are tested.

Though testing all DD-paths that are reached from the modified code is necessary, it can also miss important potential error conditions. In the example shown in Figure 3-4, assume that the assignment statement in DD-path 3 was changed from Y=Z*X to Y=Z/X. By retesting only those DD-paths reached from the modified code (Strategy 4), only a single test

case need be rerun, i.e., a test that executes DD-path 3. However, it can be seen that a fatal error occurs depending on which path was taken before the modified code. If the path containing DD-path 1 was traversed, execution continues normally. However, if the path containing DD-path 2 was executed, then a fatal error occurs at the modified DD-path because of division by zero.




| TEST CASE | DD–PATHS EXERCISED |
|---|---|
| 1 | 1, 3 |
| 2 | 2, 3 |
| 3 | 1, 4 |
| 4 | 2, 4 |

Figure 3-4. Sample Flowchart

### 3.1.5 Retest Strategy 5: Retest All DD-paths Reaching To and Reached From the Changed Code

The limitations of strategy 4 suggest the more powerful strategy of retesting all DD-paths reaching to and reachable from the modified code. In the example depicted in Figure 3-4, this would require that at least two testcases be rerun: one executing the DD-path sequence 1-3, and the other executing the DD-path sequence 2-3. This strategy, though always requiring an amount of retesting greater than or equal to that required by strategy 4, is apt to detect more errors than the strategy of retesting only those DD-paths reached from the modified code.

### 3.1.6 Retest Strategy 6: Retest All DD-paths Reaching the Changed Code and Setting Changed Data, and Reached From the Changed Code and Using Changed Data

The previous five retest strategies considered only a program's control structure when determining a retest subset. Another important element is a program's data dependency structure. The data dependency structure of a program describes the logical relationship among data elements.

Important in a discussion of data dependency are the concepts of setting data and using data. Setting a data value means to assign (or reassign) a value to that data's storage location. For example, the variable X is being set in the FORTRAN assignment shown below:

```
X=Y*2
```

Using a data value means to access a data item's storage location and utilize its value in comparing or computing some other value. The variable Y is being used in the FORTRAN statement shown below:

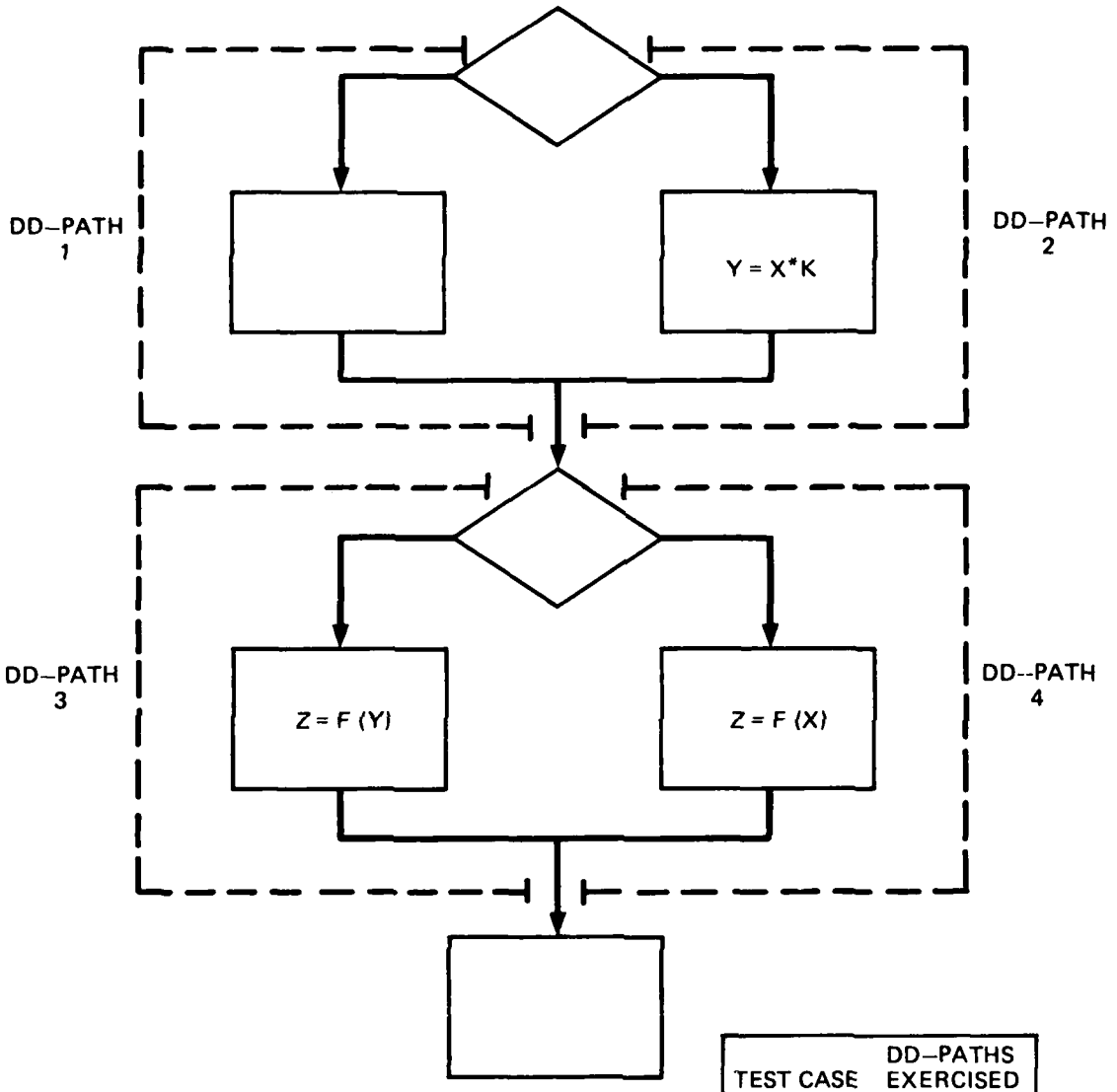
```
X=Y*2
```

It is also possible for a data value to be both set and used in the same statement. The FORTRAN assignment statement shown below is one such example:

```
X=X+1
```

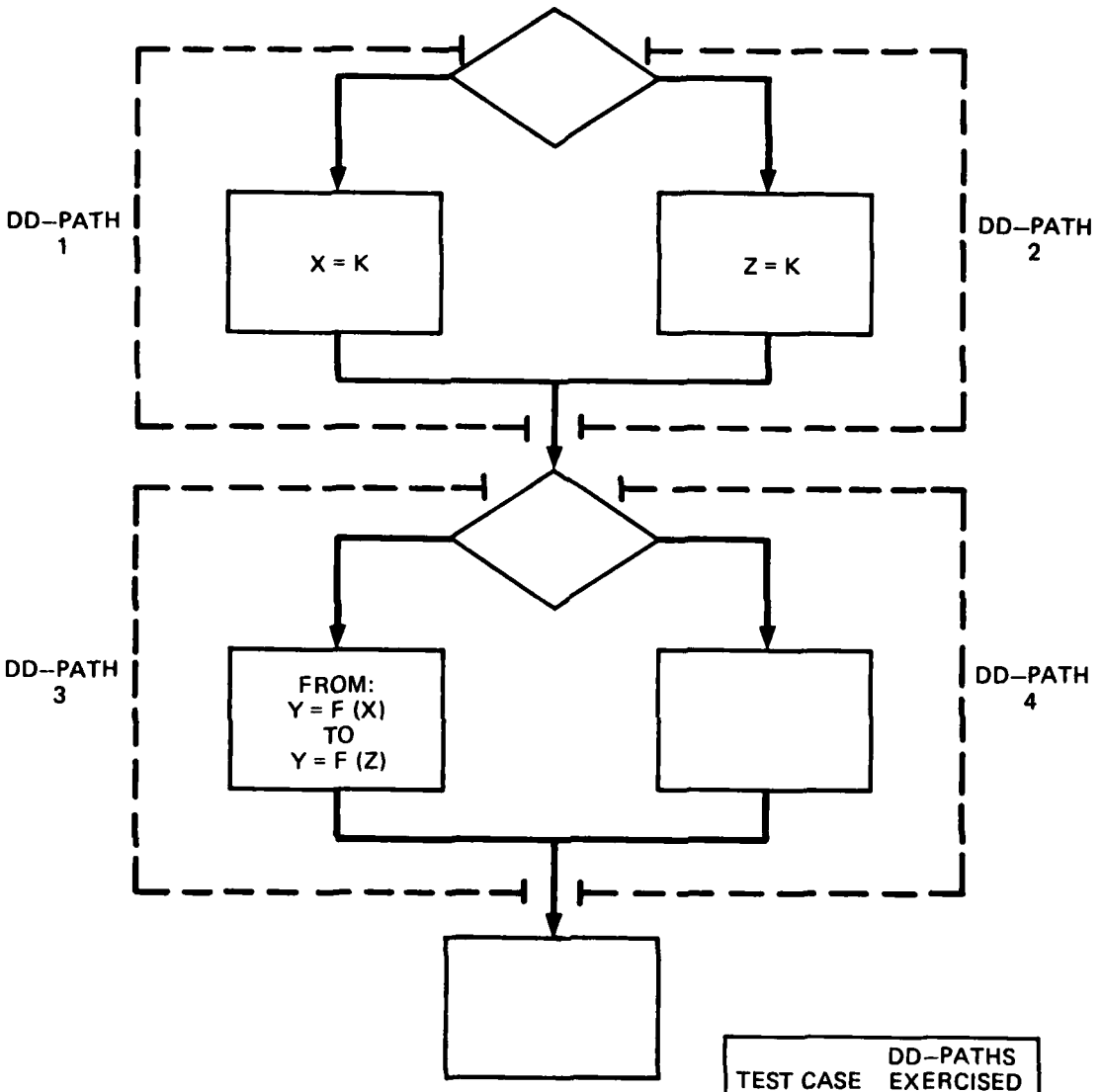
The concept of data dependency is important to the retest problem because studying a program's data logic as well as its control logic can eliminate much unnecessary testing. The example in Figure 3-5 shows how considering data logic can eliminate some of what the previous retest strategies would consider mandatory testing. For this example, assume that a constant is changed in DD-path 2 from one value to another. Clearly, this change impacts the assigned value Y, and the effects of this change should be evaluated throughout the program. Using control structure alone, as in strategies 4 and 5, a test case exercising DD-path 3 and a test case exercising DD-path 4 would be selected for retest. Introducing data dependency eliminates the need for retesting a test case exercising DD-path 4 because although control can transfer from DD-path 2 to DD-path 4, the data generated in DD-path 2 is not used in DD-path 4. Since DD-path 4 is not impacted, a test case exercising DD-path 4 would not be selected for retest. This example indicates that the minimal retest set satisfying strategy 6 will always be a subset of the minimal retest set satisfying strategy 5.

In addition to retesting test cases containing DD-paths which use variables set in the modified code, it may also be desirable to retest test cases containing DD-paths which transfer to the modified code. However, retesting test cases containing DD-paths that transfer to the modified code may be testing more than is necessary. The example in Figure 3-6 illustrates this point. Without considering data flow, strategy 5 would have dictated that a test case exercising




| TEST CASE | DD–PATHS EXERCISED |
|---|---|
| 1 | 1, 3 |
| 2 | 1, 4 |
| 3 | 2, 3 |
| 4 | 2, 4 |

Figure 3-5. Sample Flowchart




| TEST CASE | DD–PATHS EXERCISED |
|---|---|
| 1 | 1, 3 |
| 2 | 2, 3 |
| 3 | 1, 4 |
| 4 | 2, 4 |

Figure 3-6. Sample Flowchart (Strategy 6)

DD-path 1 and a test case exercising DD-path 2 be selected for retested. Since the value of X (DD-path 1) has no impact on the execution of the modified DD-path 3, a test case exercising DD-path 1 need not be selected for retest.

The previous analysis considered only data dependency of the first order. The order of data dependency refers to the dependency of one variable upon another. In Figure 3-7, the first order data dependency would show that variable E is an input to (or is used in) statement 50. The second order data dependency would show that variables C, D, and E are input to statement 50 (since D and C set E), and the third order data dependency would show that variables A, B, C, D, and E are input to statement 50. A complete effect of all orders of data dependency is called the $n^{th}$ order dependency. Algorithms which determine these dependencies are described in paragraphs 3.2.6.1 and 3.2.6.2.

```
10  INPUT A,B
20  C=A+5
30  D=B*2
40  E=D-C
50  F=E+2
```

Figure 3-7. Sample Source Code

## 3.2 DETAILED DESCRIPTION OF RETEST STRATEGIES

This section identifies the procedures and mathematical methods necessary to implement the six retest strategies. These strategies are listed numerically in Table 3-1 and will be referred to by number instead of by name throughout this section.

Table 3-1. Retest Strategies

1. Rerun all test cases.
2. Retest all testable paths through the changed code.
3. Rerun all test cases that execute the changed code.
4. Retest all DD-paths reached from the changed code.
5. Retest all DD-paths reaching to and reached from the changed code.
6. Retest all DD-paths reaching the changed code and setting changed data, and reached from the changed code and using changed data.

### 3.2.1 Strategy 1

When using the retest strategy of rerunning all test cases, the selection process is trivial. One merely reruns all test cases that were executed during program validation and compares their output with the system software specification.

### 3.2.2 Strategy 2

Because a program may contain an extremely large number of paths, strategy 2 (as described in paragraph 3.1.2), was dropped from consideration as a viable retest strategy. Therefore, procedures for its implementation are not described.

### 3.2.3 Strategy 3

Before identifying selection methods for strategy 3, it is necessary to introduce the test case cross reference matrix. Given that one has used a test monitor tool such as the JOVIAL J73 Automated Verification System (1), this matrix can be automatically constructed with rows corresponding to program DD-paths and columns corresponding to test cases. To illustrate the construction of this matrix, the test case cross reference matrix depicted in Table 3-2 was generated based on the flowchart shown in Figure 3-8.

Table 3-2. Sample Test Case Cross Reference Matrix

| DD-path | Test Case | | | | | |
|---|---|---|---|---|---|---|
| No. | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 0 | 1 | 0 | 1 |

1. Gannon, C. and R. F. Else, "JOVIAL J73 Automated Verification System User's Manual," General Research Corporation, July 1981.

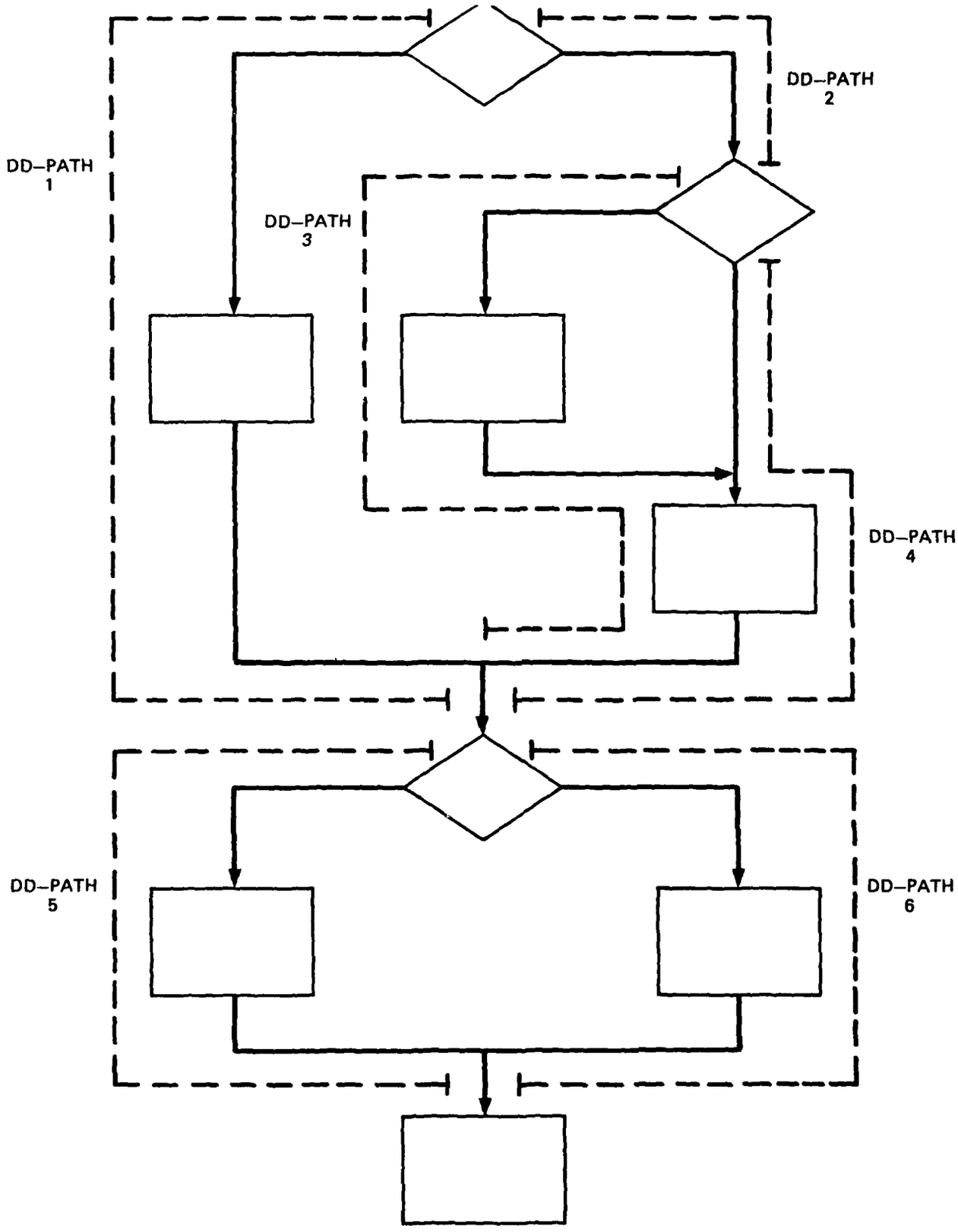



Figure 3-8. Sample Application Flowchart and Test Paths

The selection method for strategy 3 is simple, once the test case cross reference matrix is generated. One selects for retest those test cases identified by a "1" in the row of the test case cross reference matrix corresponding to the modified DD-path. For example, if DD-path 3 in the flowchart shown in Figure 3-8 is changed, the third row of the test case cross reference matrix shows that the elements in columns 3 and 4 are set to one. Therefore, according to strategy 3, test cases 3 and 4 should be rerun.

3.2.3.2 Limitations

This method restricts selection of test cases to just those test cases that execute the modified DD-path. It is not concerned with those DD-paths which reach the modified code. Because of this limitation (see page 3-9, 3rd paragraph), this strategy will no longer be considered.

3.2.4 Strategy 4

3.2.4.1 Methods and Procedures

This strategy requires the development of the test case cross reference matrix as previously discussed. In addition, DD-path reachability must be determined.

To identify DD-paths that reach the modified DD-path, reachability information is needed which can be obtained from the JOVIAL J73 Automated Verification System (1). Alternatively, reachability information can also be obtained by applying transitive closure (2) to the connectivity matrix.

---

1. Gannon, C. and R. F. Else, "JOVIAL J73 Automated Verification System User's Manual," General Research Corporation, July 1981.

2. Warshall, S., "A Theorem on Boolean Matrices," Journal of ACM, Vol. 9, No. 1 (1962), pp. 11-12.

Connectivity information can be obtained from several static analyzers such as the PACE (1).

Data from the test case cross reference matrix and reachability matrix are coupled with a optimization technique to minimize the number of test cases selected for retest.

The technique used to optimize the selection process is 0-1 integer programming. This model consists of minimizing the function:

$$Z = c_1X_1 + c_2X_2 + \ldots + c_nX_n$$

subject to the following constraints:

$$a_{11}X_1 + a_{12}X_2 + \ldots + a_{1n}X_n \geq b_1$$

$$a_{21}X_1 + a_{22}X_2 + \ldots + a_{2n}X_n \geq b_2$$

$$\vdots \qquad \vdots \qquad\qquad \vdots \quad \vdots$$

$$a_{m1}X_1 + a_{m2}X_2 + \ldots + a_{mn}X_n \geq b_m$$

$$X = 0 \text{ or } 1$$

where Z is commonly referred to as the objective function, $c_j$ is a cost element of the objective function, the $a_{ij}$ elements are coefficients of the constraints, $b_i$ is the lower bound of each constraint row i, and $X_j$ is the variable for solution which can only take on the value 0 or 1.

Let us assume that during initial validation testing of the software there were n test cases and m program DD-paths. In terms of this model, each $X_j$ corresponds uniquely to one of

---

1. Fischer, K.F., "The Product Assurance Confidence Evaluator, (PACE)," Quality Assurance Software Tools User's Guide, TRW Software Product Assurance, STP-6039, January 1977.

the n test cases. The interpretation of the final solution would be that for each j where $X_j=1$, the corresponding jth test case be included in the retest subset. For each j where $X_j=0$, the corresponding jth test case need not be included in the basic model.

The cost element of the objective function ($c_j$) is the actual cost associated with rerunning each test case. Throughout this section, we assume that the cost of rerunning each test case is identical ($c_j=1$).

The constraint coefficients ($a_{ij}$) are taken directly from the elements of the test case cross reference matrix. If testing is done with a tool that monitors testing and reports DD-path execution incidence, then the constraint coefficients can be generated automatically.

The $b_i$'s (right hand side column) reflect whether or not DD-path i needs to be tested as required by the respective retest strategy. This is determined by examining the branching structure of the program by means of the reachability matrix.

To implement this strategy one uses the kth row of the reachability matrix (where k corresponds to the modified DD-path) for the right hand side of the integer programming model. This will force a solution such that all DD-paths reachable from the modified DD-path are retested.

The optimal solution of the objective function is the minimum number of test cases necessary to ensure that all DD-paths reachable from the modified code are executed at least once. The resulting $X_j$'s, with a value of 1, identify the subset of test cases to be included in the retest package.

It is possible that the testbed does not completely test all DD-paths in a program. If a DD-path in a program is modified and the testbed does not contain a test case that

completely exercises that DD-path, then the 0-1 integer programming model cannot be solved (infeasible solution). Should this occurs, a modified solution can be reached by either eliminating the constraint(s) with all zero coefficients and resolving the model, or by constructing a new test case that executes the untested code and adding it to the model.

This basic model could be solved by a standard 0-1 integer programming algorithm. A step-by-step procedure for solving 0-1 integer programming models is presented by Taha (1). For large programs, however, the magnitude of the data could overflow available storage in many computers, thereby, preventing its practical solution. Four methods can be used to greatly reduce the size of the data needed for model formulation:

1. If a test does not execute any of the modified code, then its execution will surely not validate the modification. Therefore, one can discard from retest consideration those test cases that do not exercise the modified DD-paths. This is done by eliminating those columns (test cases) that contain a 0 in the row(s) corresponding to modified the DD-path(s).

2. One can eliminate from the model those constraints corresponding to DD-paths that are incompatible with (i.e., never reach to or are reachable from) the modified code. This is done by discarding those constraints whose $b_i$ value is 0.

3. One can eliminate from the model those constraints that are duplicates of other constraints. If one constraint is satisfied, its duplicate is also satisfied in which case the latter is redundant.

---

1. Taha, H.A. An Introduction to Operations Research, The MacMillan Company, 1971, pp. 327-341.

4. One can eliminate from the model any constraint containing all the elements included in the objective function (Z). Since any solution will satisfy that constraint, it is extraneous.

A specific example will be reviewed to illustrate the procedure. If the flow of a routine were as shown in Figure 3-8, there could be as many as 6 test cases designed to exercise all program functions. The test case cross reference matrix (Table 3-3) correlates test cases with executed DD-paths. For example, test case 1 exercises DD-paths 1 and 5. The connectivity matrix (Table 3-4), puts the logic flow of Figure 3-8 into a matrix format which can be converted into a reachability matrix (Table 3-5) by applying transitive closure. For purposes of this research, the reachability matrix has 1's in the main diagonal to assure that the modified DD-paths are retested. Let us suppose that a program modification is made in DD-path 3 (in Figure 3-8) and we apply retest strategy 4. The elements of the test case cross reference matrix (Table 3-3) serve as coefficients of the constraint expressions ($a_{ij}$'s), and the right hand side values ($b_i$'s) are taken from the third row (since DD-path 3 was modified) of the reachability matrix. The model for this sample application would be as stated in Figure 3-9.

$$
\begin{array}{lcccccccc}
\text{MINIMIZE } Z = & X_1 + & X_2 + & X_3 + & X_4 + & X_5 + & X_6 & & \\
\text{SUBJECT TO:} & X_1 + & X_2 & & & & & & \geq 0 \\
 & & & X_3 + & X_4 + & X_5 + & X_6 & & \geq 0 \\
 & & & X_3 + & X_4 & & & & \geq 1 \\
 & & & & & X_5 + & X_6 & & \geq 0 \\
 & X_1 & + & X_3 & + & X_5 & & & \geq 1 \\
 & & X_2 & + & X_4 & + & X_6 & & \geq 1
\end{array}
$$

Figure 3-9. Sample Retest Model Formulation

Table 3-3. Example Test Case Cross Reference Matrix

| | Test Case | | | | | |
|---|---|---|---|---|---|---|
| DD-path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 0 | 1 | 0 | 1 |

Table 3-4. Example Connectivity Matrix

| | DD-Path | | | | | |
|---|---|---|---|---|---|---|
| DD-Path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 3-5. Example Reachability Matrix

| | DD-Path | | | | | |
|---|---|---|---|---|---|---|
| DD-path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

The sample retest model formulation (Figure 3-9) can be reduced by the application of the reduction rules as described below:

a. Reduction Rule 1: Eliminate those columns (test cases) that contain an 0 in the row(s) corresponding to the modified DD-paths.

The third constraint in the formulation is examined to determine which test cases ($X_j$'s) exercise DD-path 3 (since DD-path 3 was modified). The third constraint indicates that only test cases 3 and 4 exercise DD-path 3, therefore, test cases 1, 2, 5, and 6 can be eliminated from all the constraints in the formulation. Figure 3-10 shows the formulation after application of reduction rule 1.

$$
\begin{array}{lll}
\text{MINIMIZE} & Z = & X_3 + X_4 \\
\text{SUBJECT TO:} & & X_3 + X_4 \geq 0 \\
 & & X_3 + X_4 \geq 1 \\
 & & X_3 \geq 1 \\
 & & X_4 \geq 1
\end{array}
$$

Figure 3-10. Sample Retest Model Formulation After Application of Reduction Rule 1

b. Reduction Rule 2: Discard all constraints whose b value is 0.

The first constraint in the reduced formulation (Figure 3-10) has a $b_i$ value of 0 and can be eliminated, further reducing the formulation as shown in Figure 3-11.

$$
\begin{array}{lll}
\text{MINIMIZE} & Z = & X_3 + X_4 \\
\text{SUBJECT TO:} & & X_3 + X_4 \geq 1 \\
 & & X_3 \geq 1 \\
 & & X_4 \geq 1
\end{array}
$$

Figure 3-11. Sample Retest Model Formulation After Application of Reduction Rule 2

c. Reduction Rule 3: Eliminate any constraint containing all 1's.

Since there are no duplicate constraints in the formulation, application of rule 3 will not further reduce the formulation.

d. Reduction Rule 4: Eliminate any constraint containing all elements included in the objective function (Z) unless it is the only remaining constraint.

In Figure 3-11, the objective function Z is equal to $x_3$ and $x_4$. The first constraint in the reduced formulation shown in Figure 3-11 contains both these elements and can be eliminated, reducing the formulation to:

$$\begin{array}{lrcl} \text{MINIMIZE } Z = & X_3 + & X_4 & \\ \text{SUBJECT TO:} & x_3 & & \geq 1 \\ & & X_4 & \geq 1 \end{array}$$

Solution of this problem using 0-1 integer programming shows that both $X_3$ and $X_4$ equal 1 and the optimal value of the objective function is 2. This means that there are two test cases to be rerun and that they are test cases 3 and 4.

3.2.4.2 Limitation

This strategy, although more comprehensive than the previous strategies, is still incomplete because it does not account for the impact of the modification to the DD-paths reached from the modified code.

3.2.5 Strategy 5

The implementation of this strategy is similar to that for strategy 4. The same matrices (connectivity, reachability, and test case cross reference) and the optimization method are required, but values for the $b_i$'s are determined differently.

3.2.5.1 Methods and Procedures

This strategy takes into account the DD-paths reaching to and reached from the modified code. To achieve this goal a

logical OR operation is performed between the Kth row and the Kth column (where K corresponds to the modified DD-path) of the reachability matrix. The result of this logical OR is used as the $b_i$'s of 0-1 integer programming model.

An example showing how to compute the $b_i$'s for 0-1 programming model based on strategy 5 is presented below. The flowchart shown in Figure 3-12, the test case cross reference matrix shown in Table 3-6, and the reachability matrix depicted in Table 3-7 are used in the formulation of this example. Once again, a modification to DD-path 3 is assumed.

| | |
|---|---|
| Row 3 of the Reachability Matrix | 0 0 1 0 1 1 |
| Column 3 of the Reachability Matrix | 0 1 1 0 0 0 |
| Logical OR | 0 1 1 0 1 1 ($b_i$'s) |

The formulation of the 0-1 integer programming model for this example is:

$$\text{MINIMIZE } Z = X_1 + X_2 + X_3 + X_4 + X_5 + X_6$$

*SUBJECT TO:*

$$\begin{aligned}
X_1 + X_2 &\geq 0 \\
X_3 + X_4 + X_5 + X_6 &\geq 1 \\
X_3 + X_4 &\geq 1 \\
X_5 + X_6 &\geq 0 \\
X_1 + X_3 + X_5 &\geq 1 \\
X_2 + X_4 + X_6 &\geq 1
\end{aligned}$$

After applying reduction rules, the model reduces to:

$$\text{MINIMIZE } Z = X_3 + X_4$$

SUBJECT TO:

$$\begin{aligned}
X_3 &\geq 1 \\
X_4 &\geq 1
\end{aligned}$$

Solution of the model indicates that test cases 3 and 4 need to be rerun.

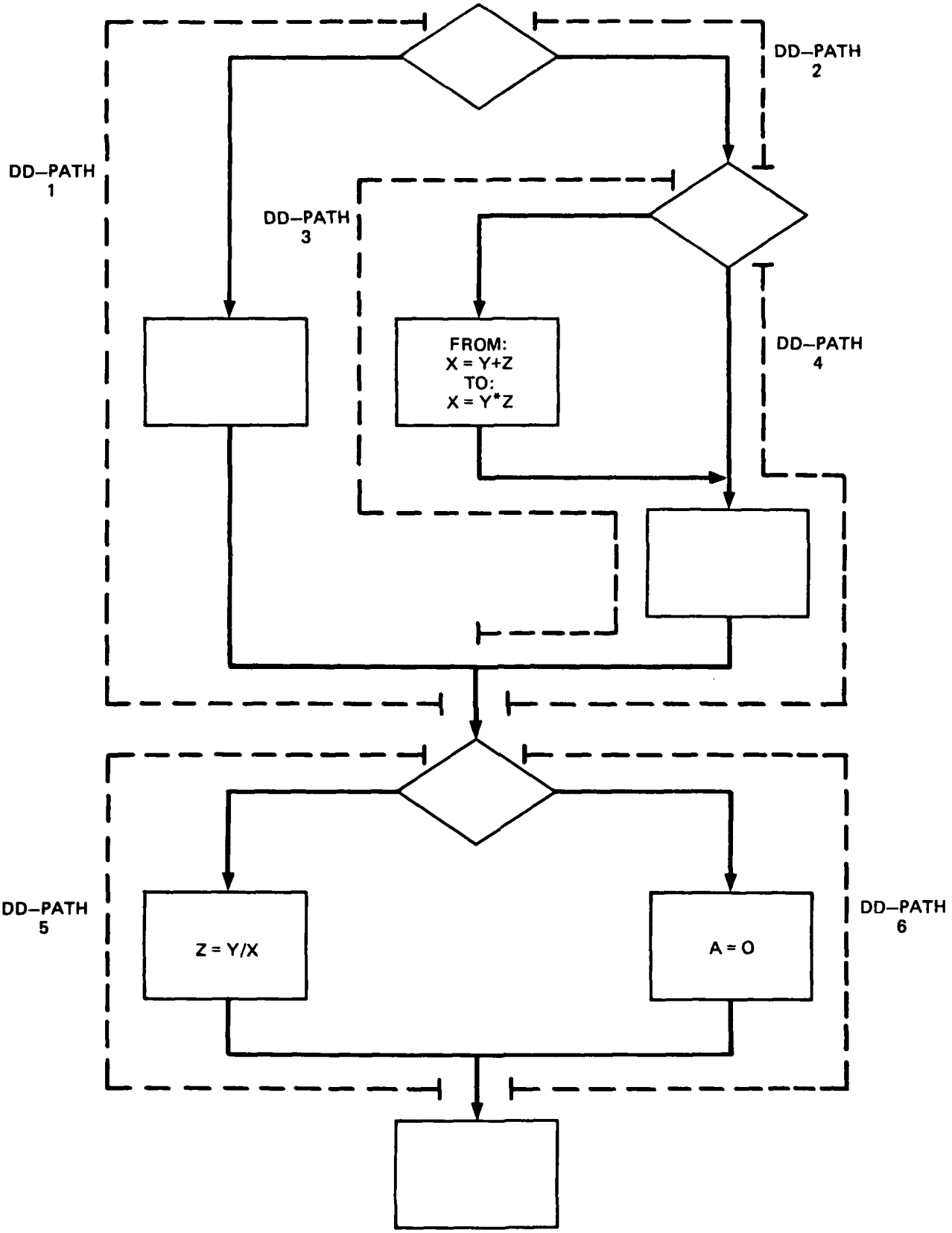



Figure 3-12. Example Formulation

Table 3-6. Example Test Case Cross Reference Matrix

| | TEST CASE | | | | | |
|---|---|---|---|---|---|---|
| DD-Path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 0 | 1 | 0 | 1 |

Table 3-7. Example Reachability Matrix

| | DD-Path | | | | | |
|---|---|---|---|---|---|---|
| DD-Path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

For this particular example, the application of strategy 5 yields the same result as the application of strategy 4. However, this is not always the case. For example, given the directed graph represented in Figure 3-12, the reachability matrix and test case cross reference matrix shown in Table 3-6 and Table 3-7 respectively, and assuming a modification to DD-path 5, the application of strategy 5 will yield a different result than strategy 4 as shown in Figure 3-13.

As indicated, the subset of the testbed selected for retest by the application of strategy 5 contains a greater number of test cases than the subset selected by the application of strategy 4. Although application of strategy 5 requires more retesting, it is more apt to detect errors.

3.2.5.2 Limitation

This strategy covers DD-paths reaching to the modified code and DD-paths reachable from the modified code. Although strategy 5 adequately considers a program's control structure, consideration is not given to a program's data dependency structure. Therefore, it may select more test cases than are necessary for retesting.

3.2.6 Strategy 6

This strategy also uses the connectivity, reachability, and test case cross reference matrices. The matrices are constructed the same way and are used for the same purposes. Once again, the values for the $b_i$'s are determined differently.

3.3.6.1 Methods and Procedures

Since this strategy is an enhanced version of strategy 5, it uses the 0-1 integer programming technique and data reduction as described before.

The critical difference here is the analysis of data dependency, which is an understanding of the flow of data

| Step | Application of Strategy 4 | Application of Strategy 5 |
|---|---|---|
| 1. Determine $b_i$'s | Row 5 of Reachability Matrix=000010($b_i$'s) | Row 5 of Reachability Matrix 000010<br>Column 5 of Reachability Matrix 111110<br>(LOGICAL OR) 111110 ($b_i$'s) |
| 2. Set up 0-1 Integer Programming Model | Minimize Z = $X_1 + X_2 + X_3 + X_4 + X_5 + X_6$<br>Subject To: $X_1 + X_2 \geq 0$<br>$X_3 + X_4 + X_5 + X_6 \geq 0$<br>$X_3 + X_4 \geq 0$<br>$X_5 + X_6 \geq 0$<br>$X_1 + X_3 + X_5 \geq 1$<br>$X_2 + X_4 + X_6 \geq 0$ | Minimize Z = $X_1 + X_2 + X_3 + X_4 + X_5 + X_6$<br>Subject To: $X_1 + X_2 \geq 1$<br>$X_3 + X_4 + X_5 + X_6 \geq 1$<br>$X_3 + X_4 \geq 1$<br>$X_5 + X_6 \geq 1$<br>$X_1 + X_3 + X_5 \geq 1$<br>$X_2 + X_4 + X_6 \geq 0$ |
| 3. Result obtained after *Application of* Reduction Rules | Minimize Z = $X_1 + X_3 + X_5$<br>*Subject To:* $X_1 + X_3 + X_5 \geq 1$ | Minimize Z = $X_1 + X_3 + X_5$<br>Subject To: $X_1 \geq 1$<br>$X_3 + X_5 \geq 1$<br>$X_3 \geq 1$<br>$X_5 \geq 1$ |
| 4. Solution to 0-1 Integer Programming Model (Test cases Selected for Retest) | Either Test case 1 or Test case 3 or Test case 5 | Test case 1 and Test case 3 and Test case 5 |

GIVEN: Directed Graph (Figure 3-12)
Test Case Cross-Reference Matrix (Table 3-7)
Reachability Matrix (Table 3-6)

ASSUMPTION: A modification to DD-path 5 has been made

Figure 3-13. Comparison of the Results of the Application of Strategy 4 vs. Strategy 5

within a program. The tool used in this analysis is the set/use matrix. Data within the set/use matrix indicates whether or not a particular variable is set or used within each DD-path. The setting of a variable occurs when a value is placed into that variable's storage location (i.e., A=5). The using of a variable occurs when the storage location of a variable is accessed and the contents read and then used (i.e., A=B). A variable can be both set and used in the same statement (i.e., X=X+1).

The cells of a set/use matrix indicate what variables are set and used within the source statements contained in DD-paths of the target program. The set/use matrix in Figure 3-14 shows the data dependency for the example flowchart in Figure 3-12. In the matrix, the occurrence of an "S", "U", or "X" indicates that a variable is set, used, or both set and used, respectively.

| | DD-paths | | | | | |
|---|---|---|---|---|---|---|
| Variables | 1 | 2 | 3 | 4 | 5 | 6 |
| A | X | U | U | U | U | U |
| B | U | U | U | U | | |
| X | | U | X | X | X | U |
| Y | | | U | | U | |
| Z | | | U | | U | |

Figure 3-14. Sample Set/Use Matrix

Given the set/use matrix and the reachability matrix, the retest subset can be reduced further than with the reachability matrix alone. Recall that for strategy 4, the row of the reachability matrix corresponding to the modified DD-path becomes the right hand side ($b_i$'s) of the 0-1 integer programming model. By reducing the number of 1's in the right hand side, the number of test cases that must be rerun may be reduced.

Two algorithms have been developed which operate on the set/use matrix and the row and column of the reachability matrix corresponding to the modified DD-paths to determine the data flow impact of the modified variables.

Analysis of the module set/use matrix can be used to further reduce the number of selected test cases. This is performed by identifying DD-paths in which the data elements can affect or be affected by the modification. The data analysis algorithm A, located in Appendix A, is used to determine all DD-paths containing data elements which potentially affect data conditions used in the modified DD-path. The data analysis algorithm B, located in Appendix B, is used to determine all DD-paths containing data elements which are potentially affected by data conditions set by the modified DD-path.

In the following example, the flow chart shown in Figure 3-15, the test case cross reference matrix illustrated in Table 3-8, the reachability matrix depicted in Table 3-9, and the set use matrix in Table 3-10 are used. A modification to DD-path 3 is assumed. The first step is to use the data analysis algorithm A to identify DD-paths containing data elements which potentially affect data conditions used in the modified DD-path. A logical AND is then performed between the result of algorithm A and column 3 (since DD-path 3 is modified) of the reachability matrix. This identifies the DD-paths which reach to and are affected by the modification. Figure 3-16 illustrates this analysis.

| | | |
|---|---|---|
| Column 3 of Reachability Matrix | 0 1 1 0 0 0 | |
| Result of Algorithm A | 1 0 1 1 1 0 | |
| Data/Logic Dependencies (Logical AND) Reaching to Modified DD-path | 0 0 1 0 0 0 | (I) |

Figure 3-16. Logical AND Operation (Step I)

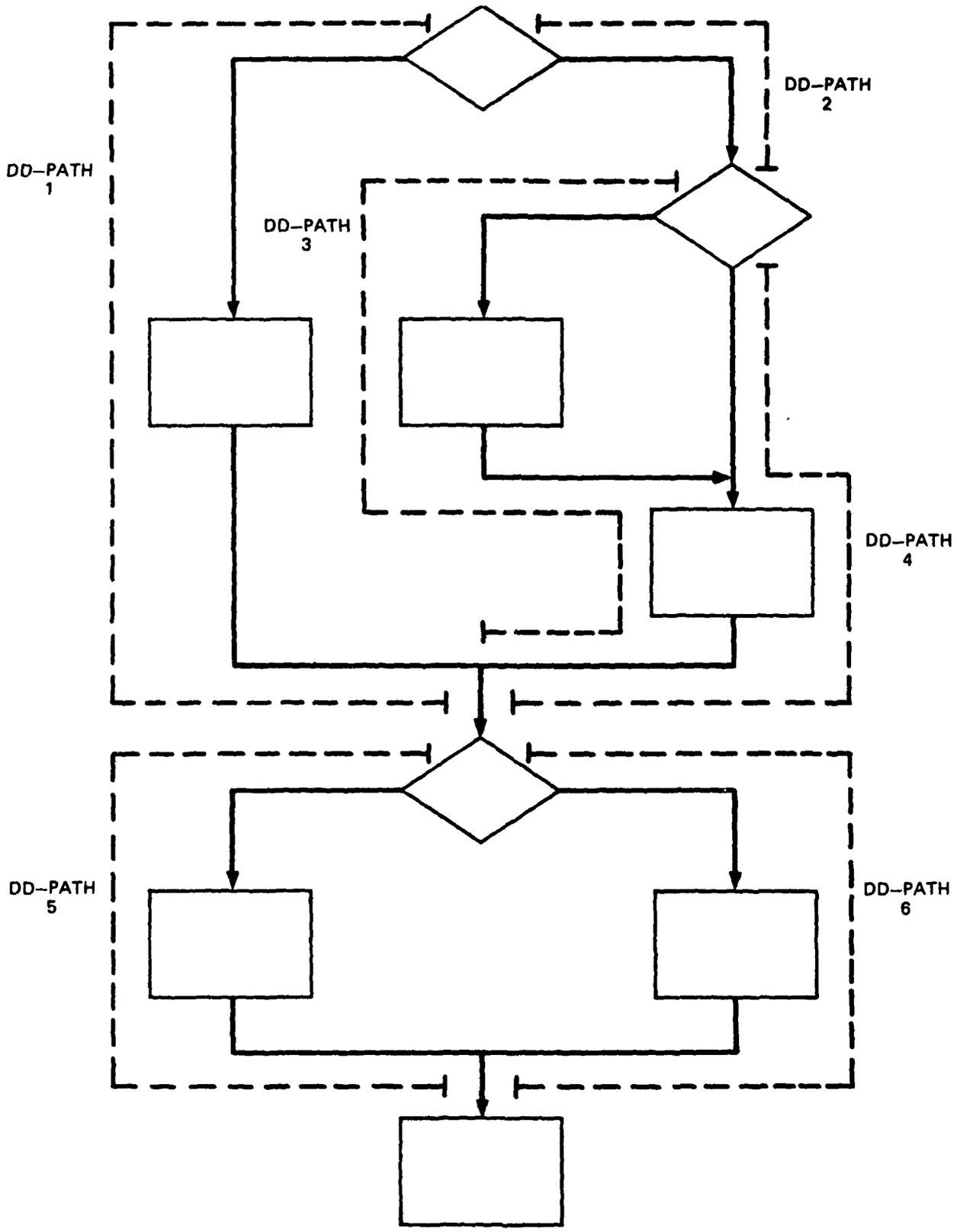



Figure 3-15. Example Formulation

Table 3-8. Example Test Case Cross Reference Matrix

| | TEST CASE | | | | | |
|---|---|---|---|---|---|---|
| DD-Path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 0 | 1 | 0 | 1 |

Table 3-9. Example Reachability Matrix

| | DD-path | | | | | |
|---|---|---|---|---|---|---|
| DD-Path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

Table 3-10. Sample Set/Use Matrix

| | DD-path | | | | | |
|---|---|---|---|---|---|---|
| Variables | 1 | 2 | 3 | 4 | 5 | 6 |
| A | X | U | U | U | U | U |
| B | U | U | U | U | | |
| X | | U | X | X | X | U |
| Y | | | U | | U | |
| Z | | | U | | U | |

The second step is to identify DD-paths reached from the modified DD-path and data elements affected by the modification. The analysis proceeds in the same manner, except the logical AND is performed between the result of the data analysis algorithm B, described in Appendix B, and the third row of the reachability matrix. Figure 3-17, demonstrates the logical AND operation used to determine the DD-paths affected by and reached from the modified DD-path.

| | | |
|---|---|---|
| Row 3 of the Reachability Matrix | 0 0 1 0 1 1 | |
| Result of Algorithm B | 0 1 1 1 1 1 | |
| Data/logic Dependencies | | |
| Reached From the Modified DD-Path | 0 0 1 0 1 1 | (II) |

Figure 3-17. Logical AND Operation (Step II)

Finally, a logical OR between the result of step (I) and the result of step (II) is performed to identify the final $b_i$'s used in the 0-1 integer programming model. Figure 3-18 illustrates this logical OR operation.

| | |
|---|---|
| Result of Step 1 | 0 0 1 0 0 0 |
| Result of Step 2 | 0 0 1 0 1 1 |
| Final $b_i$'s | 0 0 1 0 1 1 |

Figure 3-18. Logical OR Operation

The remainder of the test case selection methodology (i.e., solving the 0-1 integer programing model) is identical to that described for strategy 4 and 5.

3.2.6.2 Manual Walkthrough of Set/Use Matrix

In this paragraph, a manual walkthrough of the set/use matrix analysis using Algorithms A and B is conducted. Figure 3-19 illustrates a set/use matrix for a module containing 7 DD-paths and 3 variables.

| | DD-path | | | | | | |
|---|---|---|---|---|---|---|---|
| Variable | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| X | S | U | U | U | | | U |
| Y | | | S | S | U | U | |
| Z | | | | | | S | S |

Figure 3-19. Example Set/Use Matrix

Throughout this walkthrough, a modification to DD-path 4 is assumed.

3.2.6.2.1 Manual Walkthrough of Algorithm A

Since DD-path 4 is modified, the 4th column of the set/use matrix is flagged with a 1 in row 0 of that column and is then searched. Since row X contains U, the column array (implemented in Figure 3-20 as column 0) is flagged with a 1. The second search is of row X and column 1 is flagged in the row array (implemented in the Figure 3-19 as row 0), because the variable is set in DD-path 1. The third search is of column 1 because that column was flagged during the previous search. Since there are no variables used in DD-path 1, set in row X, or used in column 4 the algorithm terminates.

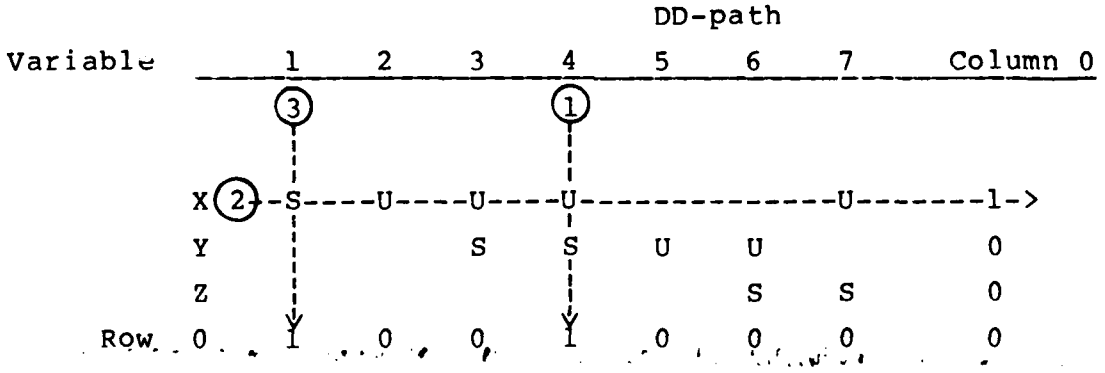



Figure 3-20. Set/Use Matrix After Applying Algorithm A

Upon algorithm termination, the columns of row 0 (which contain flags set during the algorithm's processing) containing a 1 identify DD-paths that use the variable being used in the modified DD-path. The row flags DD-paths that are affected and reach to the modification.

3.2.6.2.2 Manual Walkthrough of Algorithm B

Since DD-path 4 is modified, the fourth column of the set/use matrix is flagged with a 1 in row 0 of that column and is then searched. Since row Y contains an S, the column array (implemented in the Figure 3-17 as Column 0) is flagged with a 1. The second search is of row Y and columns 5 and 6 are flagged in the row array (implemented in Figure 3-21 as row 0,) because variable Y is used in DD-path 5 and 6. The third search is of column 5 because that column was flagged during the previous search. Since there are no variables set in that DD-path, column 6 (also flagged in search 2) is searched next. The S found in row Z of column 6 causes row Z to be flagged, which forces row Z to be searched next. Since no U's are found in row Z, the algorithm terminates.

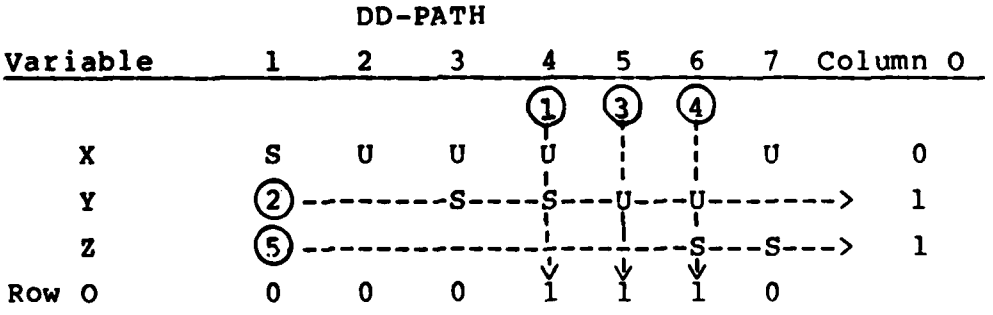



Figure 3-21. Set/Use Matrix After Applying Algorithm B

Upon algorithm termination, the columns of row 0 (which contain flags set during the algorithm's processing) containing a 1 identify DD-paths that use the variable being set in the modified DD-path. Row 0 flags DD-paths that are reached from the modification.

### 3.2.6.3 Live Example of Strategy 6

This paragraph presents a live example of the Strategy 6. The source code for the data statement module used in this example is located in Appendix E. The function of this module is to perform syntax analyses on data assignment statements written in program design language. Figure 3-22 provides a graphic representation of the data statement module and a tabular listing of statements contained in each DD-path. Each node of the graph corresponds to an executable statement in the source code. The reachability matrix and the test case cross reference matrix for the data statement module are illustrated in Table 3-11 and Table 3-12, respectively.

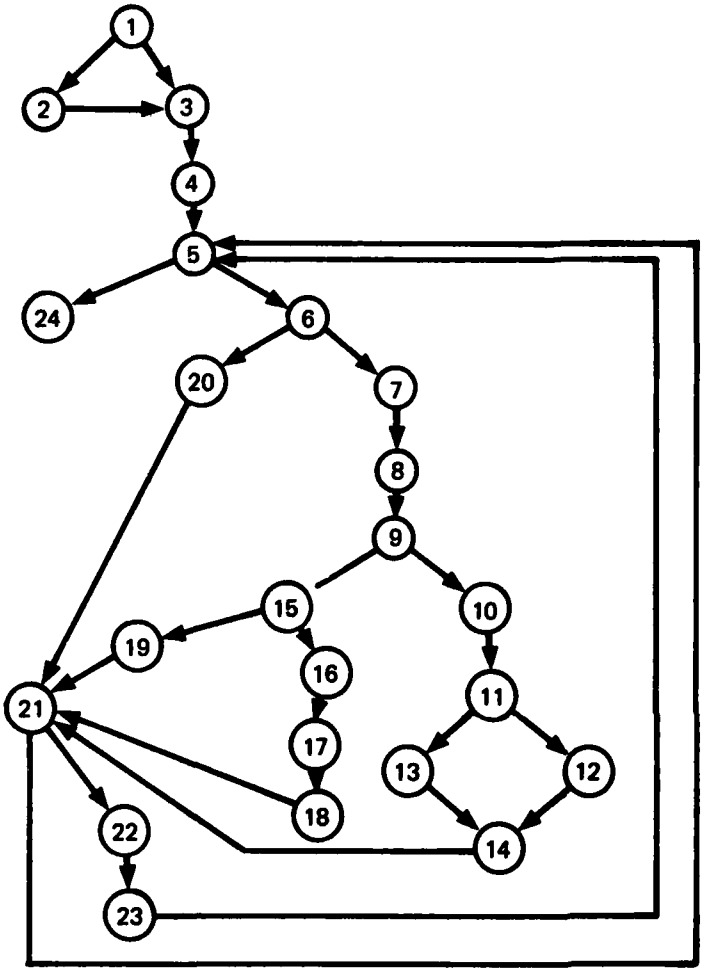
Table 3-13 illustrates the set/use matrix which shows the status of each variable in the data statement module. Tables 3-14 and 3-15 show the set/use matrix after applying Algorithm A and Algorithm B, respectively.

In this example, a modification to DD-path 5 is assumed. Therefore, we looked up column 5 of the reachability matrix (Table 3-11) to identify all DD-paths reaching DD-path 5. A logical AND is performed between the result of the analysis of algorithm A to the set/use matrix (Table 3-14) and column 5 of the reachability matrix. Figure 3-23 shows this operation.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Algorithm A | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| Col. 5 of Reachability | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Logical AND | | | | | | | | | | | | | | |
| $R_1$ | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |

Figure 3-23. Logical AND Operation for Data Statement Module

Another logical AND is performed between the result of the analysis of Algorithm B (Table 3-15) and row 5 of the reachability matrix. This operation is shown in Figure 3-24.




| DD-PATH | STATEMENTS INCLUDED IN DD-PATH |
|---|---|
| 1 | 1, 2, 3, 4, 5 |
| 2 | 1, 3, 4, 5 |
| 3 | 5, 24 |
| 4 | 5, 6 |
| 5 | 6, 7, 8, 9 |
| 6 | 9, 10, 11 |
| 7 | 11, 12, 14, 21 |
| 8 | 11, 13, 14, 21 |
| 9 | 9, 15 |
| 10 | 15, 16, 17, 18, 21 |
| 11 | 15, 19, 21 |
| 12 | 6, 20, 21 |
| 13 | 21, 22, 23, 5 |
| 14 | 21, 5 |

Figure 3-22. Graph Presentation of Data Statement Module and Identification of DD-paths

Table 3-11. Reachability Matrix for Data Statement Module

| | DD-path | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DD-path | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 11 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 12 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 13 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 14 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 3-12. Test Case Cross Reference Matrix
For Data Statement Module

| | Test Case | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DD-path | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 6 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 7 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

Table 3-13. Set/Use Matrix For Data Statement Module

| | DD-Path | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Variable | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| databit | S | S | S | | | | | | | | | | | |
| dataexist | U | | | | | | | | | | | | | |
| ind | X | X | | | X | | | | | | | | | |
| flag | X | X | U | U | | | S | S | | S | | | X | U |
| tokcode | | | | U | U | U | U | U | U | U | U | U | U | |
| ends | | | | U | | U | | | | | | U | U | |
| retcode | | | | | S | | U | U | | U | U | U | U | U |
| data | | | | | U | | U | U | U | | | | | |
| semi | | | | | | | U | U | U | U | U | | | |
| semimissg | | | | | | | U | U | | | | | | |
| datamissg | | | | | | | | | | U | | | | |
| csdp'end'of'file | | | | | U | U | | | U | U | U | U | U | |
| endmissg | | | | | | | | | | | | | U | |

Table 3-14. Set/Use Matrix For Data Statement Module after Applying Algorithm A

| | DD-path | | | | | | | | | | | | | | Col. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Variable | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 0 |
| databit | S | S | S | | | | | | | | | | | | 0 |
| dataexist | U | | | | | | | | | | | | | | 1 |
| ind | X | X | | | X | | | | | | | | | | 1 |
| flag | X | X | U | U | | | S | S | | S | | | X | U | 1 |
| tokcode | | | | U | U | U | U | U | U | U | U | U | U | | 1 |
| ends | | | | U | | U | | | | | | U | U | | 0 |
| Retcode | | | | | S | | U | U | | U | U | U | U | U | 1 |
| Data | | | | | U | | U | U | U | | | | | | 1 |
| semi | | | | | | | U | U | U | U | U | | | | 1 |
| semimissg | | | | | | | U | U | | | | | | | 1 |
| datamissg | | | | | | | | | | U | | | | | 1 |
| csdp'end'of'file | | | | | U | U | | | U | U | U | U | U | | 1 |
| endmissg | | | | | | | | | | | | | U | | 1 |
| Row 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | |

Table 3-15. Set/Use Matrix for Data Statement Module after Applying Algorithm B

| | DD-path | | | | | | | | | | | | | | Col. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Variable | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 0 |
| databit | S | S | S | | | | | | | | | | | | 1 |
| dataexist | U | | | | | | | | | | | | | | 0 |
| ind | X | X | | | X | | | | | | | | | | 1 |
| flag | X | X | U | U | | | S | S | | S | | | X | U | 1 |
| tokcode | | | | U | U | U | U | U | U | U | U | U | U | | 0 |
| ends | | | | U | | U | | | | | | U | U | | 0 |
| Retcode | | | | | S | | U | U | | U | U | U | U | U | 1 |
| Data | | | | | U | | U | U | U | | | | | | 0 |
| semi | | | | | | | U | U | U | U | U | | | | 0 |
| semimissg | | | | | | | U | U | | | | | | | 0 |
| datamissg | | | | | | | | | | U | | | | | 0 |
| csdp'end'of'file | | | | | U | U | | | U | U | U | U | U | | 0 |
| endmissg | | | | | | | | | | | | | U | | 0 |
| Row 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Algorithm B | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| Row 5 of Reachability | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Logical AND | | | | | | | | | | | | | | |
| $R_2$ | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

Figure 3-24. Logical AND Operation for Data Statement Module

To compute the final $b_i$'s, a logical OR is performed between the results of the logical operations shown in Figures 3-23 and 3-24. This logical OR is performed to determine all DD-paths reached to or reached from the modified code. Figure 3-25 illustrates this logical OR operation.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $R_1$ | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| $R_2$ | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| Logical OR | | | | | | | | | | | | | | |
| Final $b_i$'s | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

Figure 3-25. Logical OR Operation for Data Statement Module

The result of the logical OR operation is used as the $b_i$'s for the 0-1 integer programming model. Figure 3-26 illustrates the formulation of the 0-1 integer programming model for this example. To reduce the size of the model, the data reduction rules identified and described in paragraph 3.2.4.1 are applied. The reduced 0-1 integer programming model is shown in Figure 3-27.

The solution to the reduced 0-1 integer programming model identifies the minimum number of test cases necessary to implement retest strategy 6 and can be obtained by applying the step-by-step procedure described by Taha (1). Results indicate

1. Taha, H.A., An Introduction to Operations Research, The McMillan Company, 1971, p. 327-341.

$$
\begin{array}{lll}
\textbf{MINIMIZE Z=} & x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8+x_9+x_{10}+x_{11}+x_{12}+x_{13}+x_{14}+x_{15}+x_{16}+x_{17}+x_{18}+x_{19}+x_{20}+x_{21}+x_{22} & \\
\textbf{SUBJECT TO:} & x_1+x_3+x_5+x_7+x_9+x_{11}+x_{13}+x_{15}+x_{17}+x_{19}+x_{21} & \geq 1 \\
 & x_2+x_4+x_6+x_8+x_{10}+x_{12}+x_{14}+x_{16}+x_{18}+x_{20}+x_{22} & \geq 1 \\
 & x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8+x_9+x_{10}+x_{11}+x_{12}+x_{13}+x_{14}+x_{15}+x_{16}+x_{17}+x_{18}+x_{19}+x_{20}+x_{21}+x_{22} & \geq 1 \\
 & x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8+x_9+x_{10}+x_{11}+x_{12}+x_{13}+x_{14}+x_{15}+x_{16}+x_{17}+x_{18}+x_{19}+x_{20} & \geq 1 \\
 & x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8+x_9+x_{10}+x_{11}+x_{12}+x_{17}+x_{18}+x_{19}+x_{20} & \geq 1 \\
 & x_1+x_2+x_3+x_4+x_{17}+x_{18}+x_{19}+x_{20} & \geq 0 \\
 & x_1+x_2+x_3+x_4 & \geq 1 \\
 & x_{17}+x_{18}+x_{19}+x_{20} & \geq 1 \\
 & x_5+x_6+x_7+x_8+x_9+x_{10}+x_{11}+x_{12} & \geq 0 \\
 & x_5+x_6+x_7+x_8 & \geq 1 \\
 & x_9+x_{10}+x_{11}+x_{12} & \geq 1 \\
 & x_{13}+x_{14}+x_{15}+x_{16} & \geq 1 \\
 & x_1+x_2+x_5+x_6+x_9+x_{10}+x_{13}+x_{14}+x_{17}+x_{18} & \geq 1 \\
 & x_3+x_4+x_7+x_8+x_{11}+x_{12}+x_{15}+x_{16}+x_{19}+x_{20} & \geq 1
\end{array}
$$

Figure 3-26. 0-1 Integer Programming Model for Data Statement

$$
\begin{array}{llllllllllllllllll}
X_1 & + X_3 & + X_5 & + X_7 & + X_9 & + X_{11} & + X_{17} & + X_{19} & \geq 1 \\
 & X_2 & + X_4 & + X_6 & + X_8 & + X_{10} & + X_{12} & + X_{18} & + X_{20} & \geq 1 \\
X_1 & + X_2 & + X_3 & + X_4 & + X_5 & + X_6 & + X_7 & + X_8 & + X_9 & + X_{10} & + X_{11} & + X_{12} & + X_{17} & + X_{18} & + X_{19} & + X_{20} & \geq 1 \\
X_1 & + X_2 & + X_3 & + X_4 & & & & & & & & & & & & & \geq 1 \\
 & & & & & & & & & & & & + X_{17} & + X_{18} & + X_{19} & + X_{20} & \geq 1 \\
 & & & & X_5 & + X_6 & + X_7 & + X_8 & & & & & & & & & \geq 1 \\
 & & & & & & & & X_9 & + X_{10} & + X_{11} & + X_{12} & & & & & \geq 1 \\
X_1 & + X_2 & & & + X_5 & + X_6 & & & + X_9 & + X_{10} & & & + X_{17} & + X_{18} & & & \geq \\
 & & X_3 & + X_4 & & & + X_7 & + X_8 & & & + X_{11} & + X_{12} & & & + X_{19} & + X_{20} & \geq 1
\end{array}
$$

Figure 3-27. Reduced 0-1 Integer Programming Model For Data S'atement Module

that a minimum of 4 test cases must be rerun to validate the modification made to DD-path 5 and that there are 196 alternative optimum solutions (i.e., minimum combinations of different test cases that satisfy the constraints contained in this example). Table 3-16 identifies all the alternative optimums that will satisfy the 0-1 integer programming model. Given this high number of alternative optimums, some might conclude that the probability of selecting a feasible solution without benefit of the retest methodology is high. However, even if we assume that the maintenance team knows that the minimum number of test cases that must be rerun is 4, the probability, of selecting a feasible solution without further analysis is less than 3%. This is calculated by the ratio of alternative optimum solutions to the number of possible combinations of test case selections as shown in Figure 3-28. Therefore, this demonstrates the value of employing the retest methodology.

### 3.2.7 System Level Analysis

Since changes in data or logic conditions can affect code in remote locations, one concern is the global communication between system modules. This communication is accomplished via global variables shared by modules. Modification of a module within a software system could affect other modules if they use the same global variables as the modified module.

#### 3.2.7.1 Global Variable Set/Use Matrix

To determine the relationship between modules and global variables, a global variable set/use matrix is employed to monitor and record usage of global variables within the computer program. The set/use relationships between modules and global variables are defined below:

1. A variable is defined to be "set" if its value is changed (e.g., the variable A is set in the statement A=5, A=X+Y). This is represented by the letter "S" in the global variable set/use matrix.

Table 3-16. Alternative Optimum Combinations of Test Cases for Data Statement Module Modification

| | | | |
|---|---|---|---|
| (1,5,9,20) | (2,5,9,19) | (3,5,9,18) | (4,5,9,17) |
| (1,5,10,19) | (2,5,9,20) | (3,5,9,20) | (4,5,9,18) |
| (1,5,10,20) | (2,5,10,19) | (3,5,10,17) | (4,5,9,19) |
| (1,5,11,18) | (2,5,10,20) | (3,5,10,18) | (4,5,9,20) |
| (1,5,11,20) | (2,5,11,17) | (3,5,10,19) | (4,5,10,17) |
| (1,5,12,17) | (2,5,11,18) | (3,5,10,20) | (4,5,10,18) |
| (1,5,12,18) | (2,5,11,19) | (3,5,11,18) | (4,5,10,19) |
| (1,5,12,19) | (2,5,11,20) | (3,5,11,20) | (4,5,10,20) |
| (1,5,12,20) | (2,5,12,17) | (3,5,12,17) | (4,5,11,17) |
| (1,6,9,19) | (2,5,12,18) | (3,5,12,18) | (4,5,11,18) |
| (1,6,9,20) | (2,5,12,19) | (3,5,12,19) | (4,5,11,19) |
| (1,6,10,19) | (2,5,12,20) | (3,5,12,20) | (4,5,11,20) |
| (1,6,10,20) | (2,6,9,19) | (3,6,9,17) | (4,5,12,17) |
| (1,6,11,17) | (2,6,9,20) | (3,6,9,18) | (4,5,12,18) |
| (1,6,11,18) | (2,6,10,19) | (3,6,9,19) | (4,5,12,19) |
| (1,6,11,19) | (2,6,11,17) | (3,6,9,20) | (4,5,12,20) |
| (1,6,11,20) | (2,6,11,18) | (3,6,10,17) | (4,6,9,17) |
| (1,6,12,17) | (2,6,11,19) | (3,6,10,18) | (4,6,9,18) |
| (1,6,12,18) | (2,6,11,20) | (3,6,10,19) | (4,6,9,19) |
| (1,6,12,19) | (2,6,12,17) | (3,6,10,20) | (4,6,9,20) |
| (1,6,12,20) | (2,6,12,19) | (3,6,11,17) | (4,6,10,17) |
| (1,7,9,18) | (2,7,9,17) | (3,6,11,18) | (4,6,10,19) |
| (1,7,9,20) | (2,7,9,18) | (3,6,11,19) | (4,6,11,17) |
| (1,7,10,17) | (2,7,9,19) | (3,6,11,20) | (4,6,11,18) |
| (1,7,10,18) | (2,7,9,20) | (3,6,12,17) | (4,6,11,19) |
| (1,7,10,19) | (2,7,10,17) | (3,6,12,18) | (4,6,11,20) |
| (1,7,10,20) | (2,7,10,18) | (3,6,12,19) | (4,6,12,17) |
| (1,7,11,18) | (2,7,10,19) | (3,6,12,20) | (4,6,12,19) |
| (1,7,11,20) | (2,7,10,20) | (3,7,9,18) | (4,7,9,17) |
| (1,7,12,17) | (2,7,11,17) | (3,7,9,20) | (4,7,9,18) |
| (1,7,12,18) | (2,7,11,18) | (3,7,10,17) | (4,7,9,19) |
| (1,7,12,19) | (2,7,11,19) | (3,7,10,18) | (4,7,9,20) |
| (1,7,12,20) | (2,7,11,20) | (3,7,10,19) | (4,7,10,17) |
| (1,8,9,17) | (2,7,12,17) | (3,7,10,20) | (4,7,10,18) |
| (1,8,9,18) | (2,7,12,18) | (3,7,11,17) | (4,7,10,19) |
| (1,8,9,19) | (2,7,12,19) | (3,7,11,18) | (4,7,10,20) |
| (1,8,9,20) | (2,7,12,20) | (3,8,9,17) | (4,7,11,17) |
| (1,8,10,17) | (2,8,9,17) | (3,8,9,18) | (4,7,11,18) |
| (1,8,10,18) | (2,8,9,18) | (3,8,9,19) | (4,7,12,17) |
| (1,8,10,19) | (2,8,9,19) | (3,8,9,20) | (4,7,12,18) |
| (1,8,10,20) | (2,8,9,20) | (3,8,10,17) | (4,8,9,17) |
| (1,8,11,17) | (2,8,10,17) | (3,8,10,18) | (4,8,9,18) |
| (1,8,11,18) | (2,8,10,19) | (3,8,10,19) | (4,8,9,19) |
| (1,8,11,19) | (2,8,11,17) | (3,8,10,20) | (4,8,9,20) |
| (1,8,11,20) | (2,8,11,18) | (3,8,11,17) | (4,8,10,17) |
| (1,8,12,17) | (2,8,11,19) | (3,8,11,18) | (4,8,10,19) |
| (1,8,12,18) | (2,8,11,20) | (3,8,12,17) | (4,8,11,17) |
| (1,8,12,19) | (2,8,12,17) | (3,8,12,18) | (4,8,11,18) |
| (1,8,12,20) | (2,8,12,19) | (3,7,12,18) | (4,8,12,17) |

Number of Possible
Combinations of Testcases $= \frac{N!}{(N-P)!P!} = \frac{22!}{18!4!} = 7315$

N = Number of testcases in testbed
P = Minimum Number of Testcases to be rerun

Probability of Selecting A Feasible Solution $= \frac{\text{Number of Viable Solutions}}{\text{Number of Possible Combinations}} =$

$\frac{196}{7315} = .0268 = 3\%$

Figure 3-28. Probability of Feasible Testcase Selection Without Application of the Retest Methodology For Data Statement Module

2. A variable is defined to be "used" if it is accessed and its value is used during the process (e.g., variables B and C are used in the statement A=B+C). This is represented by the letter "U" in the global variable set/use matrix.

3. A variable is defined to be both "set and used" if its value is set and used in a statement (e.g., A=A+1); represented in the matrix by the letter "X".

3.2.7.2 Example Using the Global Variable Set/Use Matrix

Suppose a DD-path in another module is modified and this modification affects a global variable which has been used in a DD-path of the data statement module. The effects of this modification are reflected in the global variable set/use matrix. By scanning the global variable set/use matrix, the modules which use the modified global variable can be identified. For example, as shown in Table 3-17, if global variable A is modified in module 1, then modules 2 and 4 are also identified for further analysis. Once the modules affected by the remote modification are identified, each individual module must be analyzed to determine the specific DD-paths within the module which are affected by the remote modification. This identification of DD-paths is accomplished via the module set/use matrix because it maps variables (global, local, arguments, etc.) in a module to the DD-paths of the module. Since DD-paths affected by the remote modification are identified, the procedure for selecting test cases to be retested for these modules is accomplished by applying strategy 6.

Table 3-17. Global Variable Set/Use Matrix

| Global Variables | Modules | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| A | S | U | | U | |
| B | S | U | | | U |
| C | | U | U | | |

## 3.3 RESULT OBTAINED

### 3.3.1 Strategy Prioritization

In this section, each retest strategy is prioritized based upon the following factors:

a. Reliability. The probability that the computer program modification is fault free.

b. Application Cost. The relative cost of running the Software Retest System (SRS) for one computer program to identify those test cases that must be rerun.

c. Retest Cost. The relative cost of running the resulting test cases identified for retest by a given strategy.

d. Degree of Automation. The portion of the strategy that can be automated.

e. Cost of Implementation. The relative cost of developing an SRS based upon a given strategy.

f. Ease of Implementation. The simplicity of developing an SRS based upon a given strategy.

Each of these criterion is associated with a weight factor which identifies the importance of the criterion in meeting Air Force retest objectives. The weight factor for each criterion is given in Table 3-18. A weight factor of 3 indicates high importance in meeting Air Force retest objectives, whereas a weight factor of .5 indicates low importance.

A prioritization of the six retest strategies based on the criteria identified above is shown in Table 3-18. This prioritization is made using a scale from 1 to 5 where 5 is the most favorable and 1 is the most unfavorable. For example, a 5 would indicate high reliability, low application cost, low retest cost, high degree of automation, low implementation cost

and high ease of implementation. The last column in the table provides the total weighted cost-effectiveness for each strategy.

Since interpretation of Table 3-18 is difficult because of the polarity of the criteria, each strategy is described below.

Strategy 1: Rerun all Test Cases

The reliability of this strategy is low because it is totally dependent on the adequacy of the testbed. Though a large number of test cases may be rerun, there is no assurance that those test cases are adequate. The application cost is low because the selection procedure is trivial. The retest cost is high because all test cases need to be rerun. This strategy can be automated easily due to the simplicity of the procedure. Therefore, the implementation is simple and the cost of implementation is low.

Strategy 2: Retest All Testable Paths Through the Changed Code

Although this strategy provides very high reliability, its implementation is costly and difficult because the number of paths through a program is generally very large. Since this strategy requires the development of a test case for each path, and the number of paths through the changed code can potentially be very large; the number of test cases selected for retest may be large causing a high retest cost. Because of the problems associated with path analysis, the degree of automation is low and the application cost is high.

Strategy 3: Rerun All Test Cases That Execute the Changed Code

The reliability of this strategy is low because although all tests that execute the modified code are rerun, there is no assurance that those tests are adequate. Since the number of test cases selected for retest by this strategy is lower than for strategies 1 and 2, the retest cost was rated as moderate. The implementation of this strategy is relatively easy and cost

| Retest Strategy | Reliability/ Effectiveness (Wt=3) | Application Cost (Wt=0.5) | Retest Cost (Wt=1.5) | Degree of Automation Wt=1 | Implementation Cost Wt=1 | Implementation Ease Wt=1 | Weighted Cost Effectiveness |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 5 | 1 | 5 | 5 | 5 | 22.0 |
| 2 | 5 | 1 | 2 | 3 | 1 | 1 | 23.5 |
| 3 | 1 | 4 | 3 | 4 | 4 | 4 | 21.5 |
| 4 | 3 | 3 | 5 | 4 | 3 | 3 | 28.0 |
| 5 | 5 | 2.5 | 3 | 4 | 2.5 | 3 | 30.25 |
| 6 | 5 | 2 | 4.5 | 4 | 2 | 3 | 31.75 |

Rating Scale

1 = most unfavorable condition (i.e., low reliability, high application cost, high retest cost, low degree of automation, high implementation cost and low ease of implementation)

5 = most favorable condition (i.e., high reliability, low application cost, low retest cost, high degree of automation, low implementation cost, high ease of implementation)

Table 3-18. Retest Strategy Priority

effective since the only tools required for implementation would be a test case cross reference analysis program to identify the DD-paths covered by each test case), a statement to DD-path association table (to identify statements contained in each DD-path), and a code comparator (to determine the location of the modifications). Similarly, the application cost is relatively low because the processing required by this strategy is relatively low when compared to the other strategies. With the exception of the test case cross reference analysis program, all tools required for implementation have already been developed for other applications; therefore, the degree of automation provided by this strategy is relatively high.

Strategy 4: Retest All DD-paths Reached from the Changed Code

The reliability of this strategy is higher than the previous strategy because it provides assurance that all DD-paths reached by the modified code are covered. The retest cost is lower for this strategy than the previous strategies because the number of test cases identified for retest is lower. The application cost is slightly higher than for strategy 3 because more processing is required to determine the reachability of all DD-paths in a program. When compared to strategy 3, the implementation cost as well as the difficulty of implementation is slightly higher because a tool to determine the reachability of DD-paths within a program must be built. Since automated tools to determine the reachability of DD-paths within a program have been developed, the degree of automation provided by this strategy is the same as that provided by strategy 3.

Strategy 5: Retest All DD-paths Reaching To and Reached From the Changed Code

This strategy provides higher reliability than the previous strategy because assurance is given that all DD-paths reaching to and reached from the modified code are covered. Since all

DD-paths reaching from the modified code must be identified and logically ORed with those DD-paths reaching to the modified code, the application cost as well as the implementation cost is slightly higher for this strategy when compared to strategy 4. The retest cost is higher than for strategy 4 because the number of test cases selected for retest will be larger. The algorithm required for identification of DD-paths reaching to and reached from the modified code is straightforward, therefore, the ease of implementation is rated the same as for strategy 4.

Strategy 6: Retest All DD-paths Reaching the Changed Code and Setting Changed Data, and Reached From the Changed Code and Using Changed Data

The purpose of analyzing data dependencies within a program (strategy 6) is to reduce the amount of retesting required when only a program's control structure (strategy 5) is used as a basis for selecting test cases to retest. Therefore, strategy 6 will provide a lower retest cost than strategy 5 while maintaining the same reliability. Both the implementation cost as well as the application cost will be greater for strategy 6 than for strategy 5 because set/use analysis must be performed. Since these algorithms have already been developed and can be automated, this strategy provides the same degree of automation and ease of implementation as strategy 5.

Based on the results of the prioritization, as identified in Table 3-18, it is concluded that strategy 6 should be implemented because it is the most cost-effective of the strategies.

### 3.3.2 Methodology Characteristics

The software retest methodology has six interesting characteristics:

1. Inadequacy of Test Bed (i.e., Infeasible Solution). It is quite possible that test cases identified in a test case cross reference matrix do not completely test all

DD-paths in a program. If part of a computer program is not properly tested and that part is modified later in the maintenance phase, the 0-1 integer programming will indicate an infeasible solution and will identify the specific DD-paths that are not tested.

2. Alternative Optimum.

It may be the case that minimum combinations of <u>different</u> test cases could satisfy all constraints. Users of mathematical programming models call this an "alternative optimum". The following example is an illustration of an alternative optimum. Assume the following constraints had been derived from a problem:

$$
\begin{array}{lrcrcrcl}
\text{MINIMIZE } Z = & X_1 & + & X_2 & + & X_3 & & \\
\text{SUBJECT TO:} & X_1 & + & X_2 & + & X_3 & \geq & 1 \\
 & X_1 & + & X_2 & + & X_3 & \geq & 1 \\
 & X_1 & & & & & \geq & 1 \\
 & & & X_2 & + & X_3 & \geq & 1
\end{array}
$$

SOLUTION: Retest Set 1: $(X_1, X_2)$
Retest Set 2: $(X_1, X_3)$

The solution to this 0-1 integer programming model indicates that two minimum combinations of test cases (Retest Set 1 or Retest Set 2) exist that satisfy all constraints in the model. Therefore, selection of <u>either</u> retest set 1 <u>or</u> retest set 2 will provide complete coverage.

3. Test Cost.

Up to this point we have assumed that the cost of rerunning each test case is identical (e.g., $c_j$=1). If we apply the actual cost associated with rerunning each test case

and apply the model, the model not only will produce different sets of test cases, it will signal the most cost effective set of test cases as well.

4. Language Independency.

   The software retest methodology is language independent. This characteristic increases usability of the method during the maintenance phase. The proposed automated version of the software retest methodology carries this characteristic as well. The only language dependent need is for an automated analysis tool to provide information necessary for the preparation of the retest matrices.

5. Consistent Behavior with Different Modifications.

   The software retest methodology behaves consistently regardless of the nature of modifications made to the software system. One or a combination of the following modifications may occur:

   Addition: As long as addition of code does not create a new DD-path, a new test case is not needed. Otherwise the test case cross reference matrix should be updated (as described in the "Software Retest System Functional Description") and the retest methodology should be used to determine the impact of the new additions to the code on the rest of the software system.

   Deletion: If deleting lines of code results in the deletion of a DD-path, the test case cross reference matrix must be updated. In the maintenance phase, it is necessary to use the retest methodology after a delete operation to determine the impact of

deletion to the rest of the software especially if any global variable is involved.

Substitution: Since substitution involves both an addition and a deletion, the retest methodology must be used to determine the impact of the substitution on the program's control structure as well as its logic structure. Therefore, the test case cross reference matrix and the set/use matrices may need to be updated.

6. Alternative structural analysis options.

There are different methods that can be used to analyze the structure of a computer program. A computer program can be viewed as a set of executable statements, DD-paths, modules, or even system components. The retest methodology is flexible enough to be able to function with any option chosen by the maintainer. In this research, the DD-path option was chosen because of its compatibility with existing Air Force software.

7. Multi DD-path modification.

If modifications to more than one DD-path are made, a logical OR should be performed among the rows of the reachability matrix corresponding to the modified DD-path. This procedure should likewise be performed on the appropriate columns of the reachability matrix. The results of these operations can be used to compute the right hand side ($b_i$'s) of the 0-1 integer programming model.

# SECTION 4 - CONCLUSION

The validation of software modifications is important during both the development and the operations and maintenance phases of the software life cycle, yet a review of the software literature shows that little research has been reported in this area. This report is one of the pioneer efforts in this area, and the results reported here show promising potential for both additional theoretical research and practical applications.

The objective of this research was to investigate the feasibility of quantitative retest methods by defining alternative retest strategies, measuring their performance characteristics, and identifying implementation techniques. The defined retest strategies are:

1. Rerun all previously used test cases.
2. Retest all testable paths through the changed code.
3. Rerun all test cases which execute the changed code.
4. Retest all DD-paths reachable from the changed code.
5. Retest all DD-paths reaching to or reachable from the changed code.
6. Retest all DD-paths reaching to the changed code and setting changed data, and reached from the changed code and using changed data.

Though strategy 2 was shown to be impractical, techniques were developed to implement each remaining strategy. Implementation techniques for strategies 1 and 3 were limited to a test execution history, while implementation techniques for strategies 4, 5, and 6 were developed using more sophisticated static analysis techniques. For these strategies, the logical structure of the source code was transformed into a directed graph and the graph analyzed in terms of each strategy. The optimization technique of 0-1 integer programming was then applied to minimize the amount of retesting within the constraints inherent in each strategy.

This research has extended the state-of-the-art in software technology by:

1. Defining alternative retest strategies.
2. Building a framework for making retest decisions using a directed graph representation of the target software.
3. Applying 0-1 integer programming to the retest problem to optimize the retest solution.
4. Defining algorithms to identify data dependencies within a software module.
5. Identifying the steps necessary to automate the implementation of these techniques.

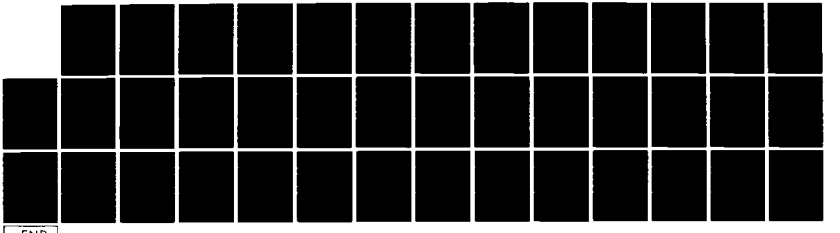
Though these advances in software technology are significant, they provide additional research opportunities in several interesting areas:

1. Additional retest strategies could be investigated which provide more reliable testing and/or reduced implementation cost.
2. Test data could be collected from sample programs and several of the six alternative retest strategies run to validate the performance and practicality of the methodology.
3. Software quality could be investigated using this model. For example, one measure of maintainability may be the number of tests cases necessary to revalidate a software modification. One could measure the performance of structured versus unstructured programs to determine the effect of program structure on maintainability.
4. The use of 0-1 integer programming could be enhanced by identifying further data reduction techniques or more efficient implementation algorithms given the structural properties of the retest model.

This research forms a solid foundation both for future research and for easing some traditional burdens of maintaining complex software systems. Implementation of retest strategy models has been demonstrated to be feasible, and tools for structuring automated retest systems have been identified. Therefore, this work can be a springboard for achieving immediate practical benefits, because the disorder of retesting software is not an inherent property, but rather stems from the typical failure of software specialists to apply the same degree of systemization to the validation of software modifications as they do to initial system validation.

AD-A123 636 SOFTWARE RETEST TECHNIQUES(U) COMPUTER SCIENCES CORP 2/2
FALLS CHURCH VA K FISCHER ET AL. OCT 82 RADC-TR-82-275
F30602-81-C-0089

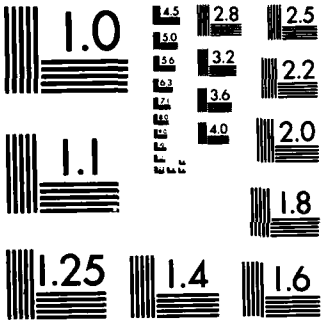
UNCLASSIFIED F/G 9/2 NL



END

FILMED

DTIC



# SECTION 5 - RECOMMENDATIONS

DoD software acquisitions are characterized by a number of attributes, which even after twenty years of trying to solve, still plague acquisition managers. One of these attributes is that the maintenance of software systems usually is more costly than their original development. This attribute is valid for both ADP and embedded computer systems. Some of the problems frequently encountered by managers of software maintenance activities are:

1. Inadequate planning.
2. Shortage of qualified personnel.
3. Lack of available tools and techniques.
4. Poor documentation.
5. Software not developed with maintainability in mind.

It is the purpose of this section to describe a proposed solution to the third problem identified above.

Though the Air Force has recognized the need for software maintenance environments (e.g., the Electronic Warfare Avionics Integration Support Facility, the F-111 Integration Support Facility, and the NORAD Off-site Test Facility), the technical personnel working within these support facilities have no guidance in the form of quantitative techniques or tools on how to retest their modified software.

Under this contract, CSC, has performed the following tasks:

1. CSC investigated existing retesting techniques and methodologies by performing a literature review and conducting on-site surveys. We found no quantitative methods that had been actually employed, and only one quantitative method (1) for which a preliminary technique was specified.

2. Based on this preliminary technique, CSC developed and described advanced techniques which could be automated to enhance existing Air Force software retesting methods. We developed six candidate retest strategies, several of which were based on a directed graph representation of the target computer program and used zero-one integer programming to minimize the amount of retesting such that all affected program elements were retested at least once. A cost-benefit analysis showed that one strategy was far superior to the others. This strategy was to rerun the minimum number of test cases such that all program components (e.g., DD-paths) reaching to and setting the changed variables and reached from and using the changed variables were tested.

3. Based on this strategy, a Functional Description document was prepared which identified and described each functional element of an automated software retest system to implement the aforementioned strategy.

During this contract, CSC has developed an excellent methodology for providing a cost effective reduction in the maintenance cost of both ADP and embedded computer software. The methodology is too complex to perform manually, and the data management too cumbersome, tedious, and error-prone. Therefore, in order to effectively implement the methodology, an automated Software Retesting System (SRS) should be developed. Such a system has already been functionally

---

1. Fischer, K. F., "A Test Case Selection Method for the Validation of Software Maintenance Modifications," Proceedings, COMPSAC '77, IEEE, November 1977, pp. 421-426.

specified in the Software Retesting System Functional Description document, and promises to be an extremely low risk effort.

To effectively satisfy the Air Force's software retesting objectives in its many varied environments, our SRS has been designed to be source language independent and highly portable from one host to another. As described in our Functional Description, the SRS consists of four functional components:

1. Retest Front-End - This component will provide the interface to the host operating environment and is the only SRS component that will require modification when moving from one host to another.

2. Command Organizer - This component provides the user interface and validates and invokes all user commands.

3. Retest Analyzer - This component performs the necessary analysis to determine the specific test cases to rerun which efficiently and reliably validate a given modification.

4. Data Base - This component indexes, stores, retrieves, and updates the necessary tables required by the Retest Analyzer.

Once the SRS is implemented and validated, CSC feels that two important steps are necessary for its acceptance by users: 1) its thorough demonstration in a real project environment, and 2) its incorporation into an integrated programming environment such as the J73 programming environment developed for RADC by CSC (under contract F30602-79-C-0051) or the Ada Integrated Environment. These last two steps are critical for system acceptance. The first step will lend credibility to the entire methodology, without which, System Program Offices will not require its use. The second step is important for technical users who will more readily accept the system if it

is embedded within their current development environment, rather than having to go to another operational environment.

We find the estimated cost for the SRS to be surprisingly reasonable. Based upon our estimate of 46,800 lines of code, the estimated level of effort for the necessary implementation steps is as follows:

| Activity | Estimated Level of Effort |
|---|---|
| Study and analysis | 10 man months |
| Specification | 10 man months |
| Implementation | 30 man months |
| Test | 12 man months |
| System Installation | 8 man months |
| Demonstration | 30 man months |
| Total | 100 man months |

Based on the comments we received during our survey of Air Force operational and support sites, the demand for a retest tool is extremely high, and the benefits are extremely worthwhile:

1. A reliable assessment of retesting effort for each modification.
2. A tool to transfer machine readable knowledge from the development environment to the maintenance environment.
3. A tool that will identify an insufficient test bed.
4. A tool to aid in the creation of new test cases.
3. A tool to identify sufficient retesting for a given software modification.
4. A methodology against which a standard can be applied.

Based on these benefits, we can only conclude that the implementation of the Software Retest System has low technical risk, is extremely cost effective, and highly beneficial to the Air Force.

## APPENDIX A

## Data Dependency Analysis Algorithm A

This algorithm is used to evaluate the set/use matrix to determine DD-paths which reach to the modified DD-paths and affect the modification.

Step 1. Set up a row array with the number of elements equal to the number of DD-paths in the target program. Set the value of each element to zero.

Step 2. Set up a column array with the number of elements equal to the number of variables in the target program. Set the value of each element to zero.

Step 3. Set the value of the element in the row array corresponding to the modified DD-path to 1.

Step 4. Scan the column of the set/use matrix corresponding to the modified DD-path, and determine the row (variable) that is being used (denoted by a "U" or an "X" in that column). In the column array, set the value of the element corresponding to the used variable to 1.

Step 5. If an element in the column array is set to 1, scan its corresponding row in the set/use matrix and set the corresponding element in the row array to 1 if that variable is set in any DD-path (denoted by a "S" or an "X" in that row). If no elements of the column array are set to 1, go to step 8.

Step 6. For each element of the row array set to 1 in step 5, scan the corresponding columns in the set-use matrix to identify what variables are used by those DD-paths (denoted by a "U" or an "X" in those columns) and set the corresponding element of the column array to 1. If no elements of the row array are set to 1, go to step 8.

Step 7. Go to step 5.

Step 8. Stop.

## APPENDIX B

## Data Dependency Analysis Algorithm B

This algorithm is used to evaluate the set/use matrix to determine DD-paths which reach from the modified DD-paths and affect the modification.

Step 1. Set up a row array with the number of elements equal to the number of DD-paths in the target program. Set the value of each element to zero.

Step 2. Set up a column array with the number of elements equal to the number of variables in the target program. Set the value of each element to zero.

Step 3. Set the value of the element in the row array corresponding to the modified DD-path to 1.

Step 4. Scan the column of the set/use matrix corresponding to the modified DD-path, and determine the row (variable) that is being set (denoted by an "S" or an "X" in that column). In the column array, set the value of the element corresponding to the set variable to 1.

Step 5. If an element in the column array is set to 1, scan its corresponding row in the set/use matrix and set the corresponding element in the row array to 1 if that variable is used in any DD-path (denoted by a "U" or an "X" in that row). If no elements of the column array are set to 1, go to step 8.

Step 6. For each element of the row array set to 1 in step 5, scan the corresponding columns in the set-use matrix to identify what variables are set by those DD-paths (denoted by an "S" or an "X" in those columns) and set the corresponding element of the column array to 1. If no elements of the row array are set to 1, go to step 8.

Step 7. Go to step 5.

Step 8. Stop.

## APPENDIX C

## BIBLOGRAPHY


1. Alford, M.W., "Software Requirements Engineering Methodology (SREM), " Proceedings, Second U.S. Army Software Symposium, Williamsburg, Virginia, October 25-27, 1978, pp. 221-234.

2. Allen, F.E. and J. Cocke, "A Program Data Flow Analysis Procedure, " Communications of ACM, Vol. 19, No. 3 (1976), pp. 137-147.

3. Belady, L.A. and M.M. Lehman, "A Model of Large Program Development," IBM Systems Journal, No. 3, 1976, pp. 225-252.

4. Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5 (1973), pp. 48-59.

5. Boehm, B.W., "Software Engineering," IEEE Trans. on Computers, Vol. C-25, No. 12, December 1976, pp. 1226-1242.

6. Brown, J.R. and K.F. Fischer, "A Graph Theoretic Approach to the Verification of Program Structures," Proceedings, Third International Conference on Software Engineering, IEEE Catalog No. 78CH1317-7C, May 1978.

7. Brown, J.R. and M. Lipow, "Testing For Software Reliability," Proceedings, International Conference on Reliable Software, IEEE Catalog No. 75CH1317-7CSR, April 1975, pp. 518-527.

8. Bucher, D.E.W., "Maintenance of the Computer Sciences Teleprocessing System," Proceedings, International Conference on Reliable Software, IEEE Catalog No. 75CH0940, April 1975, pp. 260-266.

9. Canning, R.G., "That Maintenance Iceberg," EDP Analyzer, Vol. 10, No. 10 (1972), pp. 1-14.

10. Carey, L.J., Qualifier User's Manual, Computer Software Analysts, Inc., 1974.

11. Christofides, N., Graph Theory: An Algorithmic Approach, Academic Press, 1975.

12. Clarke, L., "A System To Generate Test Data and Symbolically Execute Programs," Dept. of Computer Science, University of Colorado, Report No. CU-CS-060-75, February 1975.

13. Daly, E. B., "Management of Software Development," IEEE Transactions on Software Engineering, Vol. SE-3, No. 2 (1977), pp. 229-242.

14. Deb, R. K., "On Generation of Test Data and Minimal Cover of Directed Graphs," Proceedings of Information Processing 77, IFIP Congress, Toronto, 1977, pp. 13-16.

15. Donahoo, J. D. and D. Swearingen, "A Review of Software Maintenance Technology", RADC-TR-80-13, Rome Air Development Center, Griffiss AFB, NY, February 1980.

16. Fischer, K. F., The FORTRAN Code Auditor, Quality Assurance Software Tools User's Guide, TRW Software Product Assurance, STP-6039, January 1977.

17. Fischer, K. F., "A Test Case Selection Method for the Validation of Software Maintenance Modifications," Proceedings, COMPSAC '77, IEEE, November 1977, pp. 421-426.

18. Fischer, K. F., "The IOVAR Program," Quality Assurance Software Tools User's Guide, TRW Software Product Assurance, STP-6039, January 1977.

19. Fischer, K.F., "The Product Assurance Confidence Evaluator (PACE), Quality Assurance Software Tools User's Guide, TRW Software Product Assurance, STP-6039, January 1977.

20. Fischer, K.R., "Test Predictor-State of the Program Report," TRW Report 75-6910.05-1079, November 18, 1975.

21. Fosdick, L.D. and L.J. Osterweil, "DAVE - A Fortran Program Analysis System," Proceedings, Computer Science and Statistics: Eighth Annual Symposium on the Interface, Health Sciences Computing Facility, UCLA, February 1975, pp. 329-335.

22. Gannon, C., "Error Detection Using Path Testing and Statistic Analysis," IEEE Transactions on Computer, August, 1979.

23. Gannon, C. and R. F. Else, "JOVIAL J73 Automated Verification System User's Manual," General Research Corporation, July 1981.

24. Garfinkel, R. and G.L. Nemhauser, Integer Programming, John Wiley and Sons, Inc., 1972.

25. Gloss-Soler, S.A., The DACS Glossary - A Bibliography of Software Engineering Terms, Rome Air Development Center, Data and Analysis Center for Software, GLOS-1, October 1979.

26. Gibson, C.G. and L.R. Railing, "Verification Guidelines," TRW Software Series #71-04, August 1971.

27. Harary, F., Graph Theory, Addison - Wesley, 1971.

28. Hecht, M.S. and J.D. Ullman, "Analysis of a Simple Algorithm for Global Data Flow Problems," Proceedings, ACM Symposium on Principles of Programming Languages, 1973, pp. 207-217.

29. Hecht, M.S. and J.D. Ullman, "Graph Flow Reducibility," SIAM Journal of Computing, Vol. 1, No. 2 (1972), pp. 188-202.

30. Henley, E.J. and R.A. Williams, Graph Theory in Modern Engineering, Academic Press, 1973.

31. Hoffman, R.H., "NASA/Johnson Space Center Approach to Automated Test Data Generation," Proceedings, Computer Science and Statistics: Eighth Annual Symposium on the Interface, Health Sciences Computing Facility, UCLA, February 1975, pp. 336-341.

32. Hoffman, R. H., "The Impossible Pairs Detection Capability (IMPAIR) of the Automated Test Data Generator (ATDG)," NASA, Contract No. NAS9-14853, Houston, Texas, January 14, 1977.

33. Howden, W.E., "Methodology for the Generation of Program Test Data," IEEE Transaction on Computers, Vol. C-24, No. 5 (1975), pp. 554-559.

34. Howden, W.E., "Theoretical and Empirical Studies of Program Testing", University of Victoria, Victoria, Canada.

35. Huang, J.C., "An Approach to Program Testing," Computing Surveys, Vol. 7, No. 3 (1975), pp. 113-128.

36. King, J.C., "A New Approach to Program Testing," Proceedings International Conference on Reliable Software, IEEE Catalog No. 75CH0940-7CSR, April 1975, pp. 228-233.

37. Krause, K.W., R.W. Smith and M.A. Goodwin, "Optimal Software Test Planning Through Automated Network Analysis," Record, 1973, IEEE Symposium on Computer Software Reliability, New York, 1973, pp. 18-22.

38. Lientz, B.P. and E.B. Swanson, "Software Maintenance a User/Management Tog-of-War", Data Management, Vol. 17, No. 4 (1979), pp. 26-30.

39. Lientz, B.P., E.B. Swanson and G.E. Tompkins, "Characteristics of Application Software Maintenance," Comm. ACM, Vol. 21, No. 7, (1978).

40. Lindhorst, M.W., "Scheduled Maintenance of Applications Software," Datamation, Vol. 19, No. 5, (1973), pp. 64-67.

M., "Applications of Algebraic Methods to Computer Analysis," TRW Software Series No. 73-10, May 1973.

, M., "Some Directed Graph Methods for Analyzing uter Program," Proceedings, Computer Sciences and tistics: Eighth Annual Symposium on the Interface, alth Sciences Computing Facility, UCLA, February 1975.

Liu, C.C., "A Look at Software Maintenance," Datamation, Vol. 22, No. 11, (1976), pp. 51-55.

44. Lloyd, D. K. and M. Lipow, Reliability: Management, Methods, and Mathematics, published by the authors, Redondo Beach, California, 1977, pp 525-527.

45. Maitlen, R.L., "SURVAYOR - the Set-Use of Routine Variables Analysis Program," Applied Software Laboratory, TRW DSSG, 1975.

46. Marimont, R. B., "Applications of Graphs and Boolean Matrices to Computer Programming," SIAM Review, Vol. 2, No. 4 (1960), pp. 259-268.

47. Martin, D. E. and G. Estrin, "Path Length Computations on Graph Models of Coimputations," IEEE Transactions on Computers, Vol. C-18, No. 6 (1969), pp. 530-536.

48. McMillan Jr., C., Mathematical Programming: An Introduction to the Design and Application of Optimal Decision Machines, John Wiley and Sons, Inc., 1970.

49. Miller, R.E., "Program Testing Technology in 1980's," Proceedings of the Conference on Computing in the 1980's, IEEE, 1978.

50. Miller, E.F., RXVP: An Automated Verification System for FORTRAN, General Research Corp, Santa Barbara, CA, January 1975.

51. Mooney, J.W., "Organized Program Maintenance," Datamation, Vol. 21, No. 2 (1975), pp. 63-64.

52. Nelson, E.C., "A Statistical Basis for Software Reliability Assessment," TRW Software Series No 73-03, March 1973.

53. Ntafos, S.C. and S.L. Hakimi, "On Path Problems in Diagraphs and Application to Program Testing; IEEE Transaction on Software Engineering, Vol. SE5, No. 5, September 1979.

54. Popkin, G.S. and M.L. Shooman, "On the Number of Tests Necessary to Verify a Computer Program:, Rome Air Development Center, RADC-TR-78-229, Griffiss AFB, NY, November 1978.

55. Paige, M.R. "On Partitioning Program Graph", IEEE Transaction on Software Engineering, Vol SE-3, No. 6, November 1977.

56. Prosser, R.T., "Applications of Boolean Matrices to the Analysis of Flow Diagrams," Proceedings of the Eastern Joint Computer Conference, 1959, pp. 133-138.

57. Roy, B., "An Algorithm for a General Constrained Set Covering Problem," in Graph Theory and Computing, ed. by R.C. Reed, Academic Press, 1972.

58. Shooman, M.L. and H. Ruston, "Summary of Technical Proc ss Investigation of Software Models," Rome Air Develop nt Center, RADC-TR-79-188, Griffiss AFB, NY.

59. Shneiderman, B., Software Psychology Human Factor Computer and Information Systems, Winthrop Publisher I , 1980, p. 44.

60. Sloan, N.J.A., "On Finding the Paths Through a Network," The Bell System Technical Journal, Vol. 51, No. 2 (1972), pp. 371-390.

61. Stucki, L.G., "Tools Lessons Learned-New Strategies," McDonnell Douglas Astronautics Company, Huntington Beach, California.

62. Swanson, E.B., "The Dimensions of Maintenance," Proceedings, Second International Conference on Software Engineering, IEEE Catalog 76CH1125-4C, October 1976, pp. 492-497.

63. Taha, H.A., An Introduction to Operations Research, The MacMillan Company, 1971.

64. Tai, K., "Program Testing Complexity and Test Criteria," IEEE Transaction on Software Engineering, Vol. SE-6, No. 6, November 1980.

65. Teichroew, D., "ISDOS and Recent Extensions," Proceedings of the Symposium on Computer Software Engineering, Polytechnic Press (1976), p. 79.

66. Voges, U., Gmeiner, and Amscher, "SADAT, an Automated Testing Tool", IEEE Transaction on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 286-290.

66. Warshall, S., "A Theorem on Boolean Matrices," Journal of ACM, Vol. 9, No. 1 (1962), pp. 11-12.

67. Yau, S.S., and J.S. Collofello, Performance Considerations in the Maintenance Phase of Large-Scale Software System, Rome Air Development Center, RADC-TR-79-129, Griffiss Air Force Base, N.Y., June 1979.

68. Yau, S.S. and J. Collofello, "Some Stability Measures for Software Maintenance," IEEE Transaction Software Engineering, Vol. SE-6, No. 6, November 1980.

69. Report to the Congress of the United States, "Federal Agencies' Maintenance of Computer Program: Expensive and Undermanaged", February of 1981.

# APPENDIX D

This appendix contains a technical paper describing the retest methodology developed under this contract. The technical paper entitled, "A Retest Methodology for Modified Software," was presented at the National Telecommunications Conference '81 and was published in the conference proceedings. The National Telecommunications Conference '81 was sponsored by the Institute of Electrical and Electronic Engineers and Bell Laboratories.

# A RETEST METHODOLOGY FOR MODIFIED SOFTWARE


## ABSTRA ?

This paper describes a methodology for software retesting that leads to the development of tools which will decrease the high cost associated with current maintenance practices, as well as increase the reliability of modifications made to a software systems. Frequent modification of user requirements and/or the continuous repair of observed program errors have made the maintenance phase of the software life cycle one of the most important and often the most expensive. A major concern during this phase is the potential proliferation of errors throughout the system caused by the modification of programs. The lack of available tools and techniques forces most software maintainers to use ad hoc retesting methods which provide little, if any, quantitative information as to their test sufficiency.


## I. INTRODUCTION*

The development of computer software usually goes through an evolutionary life cycle beginning with the establishment of an operational requirement and ending with the deployment and operation of the software system. In the past, short cuts have been taken on many software projects during the early life cycle phases in order to get a product into the field quickly. This situation normally leads to decreased reliability and extremely expensive subsequent modifications during the maintenance phase. Unfortunately, most software systems take this path causing frequent modifications and updates. One problem area in the maintenance phase, called retest, arises

---


* This work was supported by the U.S. Air Force Rome Air Development Center under contract F30602-81-C-0089.

when attempting to revalidate the system due to code modifications or code additions. Retest is the act of rerunning certain tests to verify a change to an existing system. It differs from the test activity, which is concerned with planning and executing tests that initially validate the entire software system. Retest answers the following questions:

- For any given modification, what other section(s) of the software is impacted by that modification?
- For the identified section(s) of code that could be affected, what test cases should be rerun to assure the proper execution of existing capability?

The decision of what to retest and how thoroughly to do so is a major problem for software managers and researchers and has not yet been adequately resolved. In research, little work has been done in the area of retest methodologies [1]. The purist will demand that all previously used test cases be rerun. The pragmatist will leave the decision to the discretion of the test director as he believes the test director knows the software best, and by using engineering judgment and his knowledge of the code he often manually selects the subset of previously completed test cases to be rerun. Other retest methods may be: to rerun a number of randomly selected test cases; to rerun all test cases that execute the modified code; or to execute a new set of test cases that exercise all the program's major capabilities to give the user "confidence" (though not statistically) that the software operates properly [2].

Each method has some beneficial attributes, yet none gives a completely reliable solution. Rerunning all previously used test cases is almost always impractical as validation tests for large computer programs may number in the hundreds. The test director may be able to select for retest those tests that address the functional modifications, but he may not be aware

that modified data conditions could cause execution of non-functional paths resulting in inaccurate computation that may go undetected for years. What is needed is a quantitative method for assuring that new program modifications do not introduce new errors into the code. To prove this would require an analysis of every program path, but this has been formally shown to be a difficult task in all but the most trivial cases. Though the need for retesting can arise during both the testing phases of development and the operations and maintenance phase [2], this paper will discuss retest only in the context of the operations and maintenance phase since more life cycle costs are spent in this phase rather than in the testing phase.

Section II of this paper presents an overview of the methodology. An example of applying the methodology is presented in Section III. Section IV discusses data dependency.

## II. OVERVIEW

Since changes in data or logic conditions can affect code in remote locations, one concern is the global communication between system modules. This communication is accomplished via global variables shared by modules. Modification of a module within a software system could affect other modules if they use the same global variables as the modified module. To determine the relationship between modules and global variables, a global variable set/use matrix is employed to monitor and record usage of global variables within the software system. The set/use relationships between modules and global variables are defined below:

- A variable is defined to be "set" if its value is changed (e.g., the variable A is set in the statement A=5, A=X+Y). This is represented by the letter "S" in the global variable set/use matrix.
- A variable is defined to be "used" if it is accessed and its value is used during the process (e.g., variables B and

C are used in the statement A=B+C). This is represented by the letter "U" in the global variable set/use matrix.

- A variable is defined to be both "set and used" if its value is set and used in a statement (e.g., A=A+1); represented in the matrix by the letter "X".

At the module level, the focus is on structural elements called segments and the interrelationship of segments within a module. A segment is defined as a continuous sequence of executable statements with only one entry point at the beginning and one exit point at the end. By executing the first statement in any segment, all other statements in that segment are also executed [2]. The retest methodology employs knowledge of the reachability among segments. The reachability matrix (shown later) identifies, for each segment, all the other segments that can reach to it or be reached from it (either directly or indirectly). Two other matrices of importance are the module level set/use matrix which identifies the status of all local variables, global variables, and arguments within modules and the test case cross reference matrix which identifies what segments are tested by each test case. Both of these matrices are further discussed in subsequent sections of this paper.

The selection of the optimal subset of test cases is accomplished by using the 0-1 integer programming technique with data provided by the reachability, set/use, and test case cross reference matrices. The following is an abstract formulation of the 0-1 integer programming model [4] consisting of minimizing the function.

$$Z = c_1X_1 + c_2X_2 \ldots + c_nX_n$$

subject to the following constraints

$$\begin{aligned}
a_{11}X_1 + a_{12}X_2 \ldots + a_{1n}X_n &\geq b_1 \\
a_{21}X_1 + a_{22}X_2 \ldots + a_{2n}X_n &\geq b_2 \\
\vdots \quad\quad \vdots \quad\quad \vdots & \\
a_{m1}X_1 + a_{m2}X_2 + \ldots + a_{mn}X_n &\geq b_m \\
X_j = 0 \text{ or } 1 &
\end{aligned}$$

where $c_j$ is the cost element for running each test case (which we assume is one), $a_{ij}$ is an element of the constraint coefficient matrix, and $b_i$ is the lower bound of each constraint row i. The variable for solution, $X_j$, corresponds to $j^{th}$ test case in the test case cross reference matrix. M and n are the number of segments and number of test cases associated with each module, respectively. The constraint coefficient matrix ($a_{ij}$) is taken directly from the test case cross reference matrix, and the right hand side ($b_i$) is taken from the logical OR of the applicable rows and columns of the reachability matrix corresponding to the modified segment. The exact procedure is explained via example in the next section.

From the solution of this model, the value of the objective function Z will give the minimum number of test cases necessary to assure full retest coverage, and the values of $X_j$ that are equal to one will identify the specific test cases which form the optimal retest subset.

## III. EXAMPLE MODEL FORMULATION

Retest methodology views a software module as a directed graph where each node is a segment and the arcs represent connectivity between segments. The directed graph for the module used as an example in this section is shown in Figure 1.

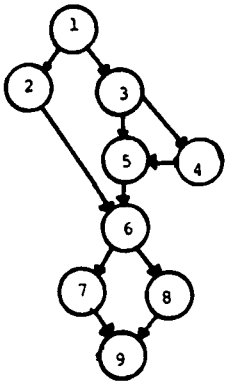



Figure 1. Directed Graph Presentation of a Module

Associated with each module is a connectivity matrix and a reachability matrix. The connectivity between the segments in Figure 1 is shown in the connectivity matrix in Figure 2. Whenever segment i is connected to segment j, the connectivity matrix element (i,j) is 1; 0 otherwise. The reachability matrix, which is the transitive closure of the connectivity matrix, depicts the reachability of segments from each other [5]. If a segment can reach another segment in the module, directly or indirectly, element (i,j) of the reachability matrix is 1; 0 otherwise. For purposes of this research we assume every segment reaches itself, and therefore all elements of the main diagonal are set to 1. Figure 3 illustrates the reachability matrix respectively based on the directed graph representation of the module shown in Figure 1.

| | | TO | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| FROM | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | 3 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 4 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 2. Connectivity Matrix

| | | TO | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 2 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 4 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| FROM | 5 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| | 6 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Figure 3. Reachability Matrix

Additionally, a test case cross reference matrix is associated with each module. This matrix is constructed by identifying segments executed by test cases during the testing phase of software development. If any element (i, j) of the test case cross reference matrix is 1, segment i is executed by test case j; 0 otherwise. For example, in Figure 4, test case 1 exercises segments 1, 2, 6, 7 and 9 of the example module. Column 2 through 6 of the test case cross reference matrix correspond to the other test cases used.

| | Test Cases | | | | | |
|---|---|---|---|---|---|---|
| Segments | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 | 1 | 1 |
| 5 | 0 | 0 | 1 | 1 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 0 | 1 | 0 | 1 | 0 |
| 8 | 0 | 1 | 0 | 1 | 0 | 1 |
| 9 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 4. Test Case Cross Reference Matrix

Let us assume that the module contains three variables: X, Y, and Z represented by the set/use matrix in Figure 5.

| Variables | Segments 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| X | S | U | U | U | | | U | S | |
| Y | | | S | S | U | U | | | S |
| Z | | | | | | U | S | S | |

where:
S: variable i set in segment j
U: variable i used in segment j
X: variable i set and used in segment j

Figure 5. Module Set/Use Matrix.

In this example, we assume a change to segment 2 is made. The following analysis should be performed to select the test cases for retest. The nine constraint expressions (i.e., one for each segment), corresponding to rows of the test case cross reference matrix, serve to assure that at least one test case executes every segment that is reached from or reaches to the modified segment. The right hand values ($b_i$'s) are the result of a logical OR operation performed between the row and column of the reachability matrix associated with modified segment (in this case row and column two). The outcome is incorporated into the 0-1 integer programming model as the right hand side ($b_i$'s) values.

$$\text{MINIMIZE} \quad Z = X_1 + X_2 + X_3 + X_4 + X_5 + X_6$$

$$\text{SUBJECT TO} \quad X_1 + X_2 + X_3 + X_4 + X_5 + X_6 \geq 1$$

$$X_1 + X_2 \geq 1$$

$$X_3 + X_4 + X_5 + X_6 \geq 0$$

$$X_5 + X_6 \geq 0$$

$$X_3 + X_4 + X_5 + X_6 \geq 0$$

$$X_1 + X_2 + X_3 + X_4 + X_5 + X_6 \geq 1$$

$$X_1 + X_3 + X_5 \geq 1$$

$$X_2 + X_4 + X_6 \geq 1$$

$$X_1 + X_2 + X_3 + X_4 + X_5 + X_6 \geq 1$$

Figure 6. 0-1 Integer Programming Model

This model can be reduced by several reduction methods [2,4].

$$
\begin{aligned}
\text{Minimize } Z = {} & X_1 + X_2 \\
\text{Subject to } & X_1 + X_2 \geq 1 \\
& X_1 + X_2 \geq 1 \\
& X_1 \geq 1 \\
& X_2 \geq 1 \\
& X_1 + X_2 \geq 1
\end{aligned}
$$

After the reduction process, redundant constraints may appear and may be removed as shown in this example. The final model formulation reduces to:

$$
\begin{aligned}
\text{Minimize } Z = {} & X_1 + X_2 \\
\text{Subject to } & X_1 \geq 1 \\
& X_2 \geq 1
\end{aligned}
$$

Solution of this example shows that both $X_1$ and $X_2$ equal 1 and the optimal value of the objective function is 2. This means that there are two test cases to be rerun and that they are test cases 1 and 2.

## IV. DATA DEPENDENCY

In the previous section, the methodology selected test cases based solely on a module's logic structure. However, analysis of data dependency can be performed to further reduce the number of test cases selected for retest. Previously the global variable and module level set/use matrices were briefly mentioned, however, it is now necessary to fully describe their purposes and utilization. The module set/use matrix reflects the status of each global variable, parameter, argument, and local variable. Global variables are included in the matrix to facilitate the identification of segments in which the global variables are used. Arguments are included for invocation of those modules whose arguments change during the modification. The set/use matrix serves two purposes. First, it can reduce the number of selected test cases associated with the module in

which the modification was made. Secondly, it identifies test cases that need to be rerun as a result of modification to other modules. The following two examples illustrate each purpose and utilization of the set/use matrix.

Analysis of the module set/use matrix can be used to reduce the number of selected test cases. This analysis is performed by using the algorithms described in Appendices A and B to identify segments in which the data elements can affect or be affected by the modification. Algorithm A is used to determine all segments containing data elements which potentially affect data conditions used in the modified segment. Algorithm B is used to determine all segments containing data elements which are potentially affected by data conditions set by the modified segment. In this example, the first step is to use the algorithm described in Appendix A. A logical AND is then performed between the result of the algorithm and column two (since segment 2 is modified) of the reachability matrix. This identifies the segments which reach to and affect the modification of segment two. Figure 6 illustrates this analysis.

| | |
|---|---|
| Column two of reachability matrix | 1 1 0 0 0 0 0 0 0 0 |
| Result of the Algorithm A | 1 1 0 0 0 0 0 0 0 0 |
| Logical AND | ——————— |
| Data/logic dependencies reaching to modified segment (I) | 1 1 0 0 0 0 0 0 0 0 |

Figure 7. Logical AND Operation

The second step is to identify segments reached from the modified segment and data elements affected by the modification. The analysis proceeds in the same manner, except the logical AND is performed between the result of the algorithm described in Appendix B and the second row of the

reachability matrix. Figure 8, demonstrates the logical AND operation to determine the segments affected by and reached from the modified segment.

| | |
|---|---|
| Row two of the reachability matrix | 0 1 0 0 0 1 1 1 1 |
| Result of the algorithm B | 0 1 0 0 0 1 1 1 0 |
| Logical AND | _____ |
| | 0 1 0 0 0 1 1 1 0 (II) |

Figure 8. Logical AND Process

Finally, a logical OR between the result of step one and the result of step two is performed to identify the final $b_i$s used in the 0-1 integer programming model. Figure 8 illustrates this logical OR operation.

| | |
|---|---|
| Step one results (I) | 1 1 0 0 0 0 0 0 0 |
| Step two results (II) | 0 1 0 0 0 1 1 1 0 |
| Logical OR | _____ |
| Final $b_i$'s | 1 1 0 0 0 1 1 1 0 |

Figure 9. Logical OR Operation

The next example illustrates the use of the set/use matrix to identify test cases that need to be rerun as a result of modification to a remote module. Suppose a segment in another module is modified and this modification affects a global variable which has been used in a segment of the example module. The effects of this modification are reflected in the global variable set/use matrix. By scanning the global variable set/use matrix, the modules which use the modified global variable can be identified. For example, as shown in

Figure 10, if global variable A is modified in module 1, then modules 2 and 4 are also identified for further analysis. Once the modules affected by the remote modification are identified, each individual module must be analyzed to determine the specific segments within the module which are affected by the remote modification. This identification of segments is accomplished via the module set/use matrix because it maps variables (global, local, arguments, etc.) in a module to the segments of the module. Since segments affected by the remote modification are identified, the procedure for selecting test cases to be retested is the same as the procedure described in example 1.

| | Modules | | | | | |
|---|---|---|---|---|---|---|
| Global Variables | | m1 | m2 | m3 | m4 | m5 |
| | A | S | U | | U | |
| | B | S | U | | | U |
| | C | | U | U | | |

Figure 10. Global Variable Set/Use Matrix

## V. CONCLUSION

Software Retesting is important during both the development phase and the operations and maintenance phase of the software life cycle. However, a review of the software literature shows that little research has been reported in this area.

This paper has presented a feasible methodology for retesting modified software based on rerunning previously used test cases.

In addition, this research has extended the state-of-the-art in software technology by:

1. Building a framework for making retest decisions using a directed graph representation of the target software

2. Applying 0-1 integer programming to the retest problem to optimize the retest solution
3. Defining an algorithm to identify data dependencies within a software module.

Though these advances in software engineering technology are significant, they provide additional research opportunities in several interesting areas. Additional retest strategies should be investigated in order to provide both more reliable and more cost efficient testing procedures. Software quality should also be investigated using this model. For example, one measure of maintainability may be the number of test cases necessary to revalidate a software modification. One should also measure the performance of structured versus unstructured programs to determine the effect of program structure on maintainability. Current research is investigating the cost and effectiveness of several retest strategies against both a recently employed ad hoc method and the testing limit of rerunning all tests. Preliminary results show that performing a data dependency analysis can significantly reduce the number of tests needed to be rerun, while maintaining high confidence that the software is being adequately retested.

## APPENDIX A

An Algorithm for the evaluation of the set/use Matrix to determine segments which reach to the modified segment and affect the modification.

1. Set up a row array with the number of elements equal to the number of segments in the target program. Set the value of each element to zero.

2. Set up a column array with the number of elements equal to the number of variables in the target program. Set the value of each element to zero.

3. Set the value of the element in the row array corresponding to the modified segment to 1.

4. Scan the column of the set/use table corresponding to the modified segment, and determine the row (variable) that is being used (denoted by an "U" or an "X" in that column). In the column array, set the value of the element corresponding to the used variable to 1.

5. If an element in the column array is set to 1, scan its corresponding row in the set/use table and set the corresponding element in the row array to 1 if that variable is set in any segments (denoted by a "S" or and "X" in that row). If no elements of the column array are set to 1, go to step 8.

6. For each element of the row array set to 1 in step 5, scan the corresponding columns in the set-use table to identify what variables are used by those segments (denoted by a "U" or an "X" in those columns) and set the corresponding elements of the column array to 1. If no elements of the row array are set to 1, go to step 8.

7. Go to step 5.

8. Stop.

## APPENDIX B

An Algorithm for the evaluation of set and use Matrix to determine segments reached from the modified segment and are affected by the modification.

1. Set up a row array with the number of elements equal to the number of segments in the target program. Set the value of each element to zero.
2. Set up a column array with the number of elements equal to the number of variables in the target program. Set the value of each element to zero.
3. Set the value of the element in the row array corresponding to the modified segment to 1.
4. Scan the column of the set/use table corresponding to the modified segment, and determine the row (variable) that is being set (denoted by an "S" or an "X" in that column). In the column array, set the value of the element corresponding to the set variable to 1.
5. If an element in the column array is set to 1, scan its corresponding row in the set/use table and set the corresponding element in the row array to 1 if that variable is used in any segments (denoted by a "U" or and "X" in that row). If no elements of the column array are set to 1, go to step 8.
6. For each element of the row array set to 1 in step 5, scan the corresponding columns in the set-use table to identify what variables are set by those segments (denoted by an "S" or an "X" in those columns) and set the corresponding element of the column array to 1. If no elements of the row array are set to 1, go to step 8.
7. Go to step 5.
8. Stop.

## REFERENCES

[1] Gibson, C. G., and L. R. Railing, Verification Guidelines, TRW Document 17618-H200-RO-00, prepared for NASA/JSC under contract NAS 9-8166 August 1971.

[2] Fischer K. F., "A Test Case Selection Method for the Validation of Software Maintenance Modification", Proceedings, COMPSAC '77, IEEE, Nov. 1977.

[3] Warshall, S., "A Theorem on Boolean Matricies", Journal of ACM, IX, 1, January 1962.

[4] Davis, R. E., D. A. Kendrik, and Weitzman, Branch-and-Bound Algorithm for Zero and One Integer Programming Problems", Operation Research, Vol. 19 (1971), pp. 1036-1044.

[5] Allenson, R. E., et.al., Automated Verification System Programmer's Guide, TRW Systems, Note No. 72 FMT.

# APPENDIX E

This appendix contains the source code for the data statement module used as an example in Section 3. The function of this module is to perform syntax analyses on data assignment statements written in program design language.

```
start
!compool ('j73sio.cmp');
!compool ('cconst.cmp');
!co pool ('pdxtrl.cmp');
!compool ('pdcnst.cmp');
!compool ('pderrs.cmp');
!compool 'cstmer.cmp' csdp'end'of'file;
pet proc pdldata (:ret'code);

%*****************************************************************************%
%                                                                             %
%                                  PROLOGUE                                   %
%                                                                             %
%  AUTHOR / DATE WRITTEN: P Hall / November 29, 1980                          %
%  MODIFICATION AUTHOR(S) / DATE:                                             %
%      P. Hall (C)  / January 9, 1981.                                        %
%  ARGUMENTS:                                                                 %
%      token: (input/output) character. Sequence of characters                %
%                          to be processed.                                   %
%      token'code: (input/output) integer. representation of token type.      %
%      ret'code: (input/output) integer. status error code.                   %
%  PARENT MODULE: none.                                                       %
%  EXTERNAL MODULES:                                                          %
%    pdlscan:   To provide tokens from source file;                           %
%    pdlserver: To provide error messages and warnings.                       %
%  GLOBAL DATA STRUCTURES:                                                    %
%    ind: integer, external, indicates level of indentation.                  %
%    datadit: external boolean to indicates existence of data block.          %
% Note:  All above variables are  Defined in "pdlexterls" compool.            %
%  FUNCTIONAL NARRATIVE:                                                      %
%      Recognizes the syntax of a data block and its matching end.            %
% Since the information in the data block does not follow any specified       %
% syntax, and the type of declaration varies, this module eliminates          %
% the entire block. The only statement that is recognized by                  %
% this module is "end data;". The pdl syntax of the data block is:            %
% data <declarations and  comments> end data;                                 %
%                                                                             %
%*****************************************************************************%


begin
!copy 'pdacls.j73'; %include declaration of pdlscan, pdlexp, pdlserver%
item flag b;
if   datadit = true;
     pdlserver (dataexist);
ind = ind + 2;
flag = false;
while(flag = false);
begin
    if   tokcode = ends;
      begin
         ind = ind - 2;
         pdlscan(:ret'code);
         if  tokcode = data;
             begin
                pdlscan(:ret'code);
                if   tokcode = semi;
                     pdlscan (:ret'code);
                else pdlserver (semimiss);
```

```
                    flag = true;
                end
            else if tokcode = semi;
                    begin
                       pdlserver (datamissg);
                       pdlscan(:ret'code);
                       flag = true;
                    end
                   else pdlscan (:ret'code);
          end  % if tokcode = end %
      else pdlscan(:ret'code);
      if   ret'code = csdp'end'of'file;
           begin
              pdlserver (endmissg);
              flag = true;
           end
   end   % while %
   databit = true;
   end  % pdldata %
   term
```

# APPENDIX F

## GLOSSARY

The following terms and definitions pertinent to this document are described below. Acronyms are defined in Appendix G.

Automated Verification System (AVS) -

A software tool that performs both static and dynamic analysis to aid in the testing and verification of computer programs.

Branch -

A branch (or Decision-to-Decision path) is the ordered sequence of statements the program performs as a result of the outcome of a decision up until the evaluation of the predicate in the next decision statement encountered. Figure 1-1 provides a diagram of a branch.

Connectivity Matrix -

An NxN matrix where N equals the number of program elements (such as decision-to-decision paths), and where any (i,j) element is denoted on the matrix by a 1 if program element i transfers directly to program element j. If program element i does not transfer directly to program element j, then element (i,j) is denoted by a 0. This matrix may also be known as an incidence matrix or an adjacency matrix.

Data Dependency -

The logical connectivity or relationship of data elements within a program.

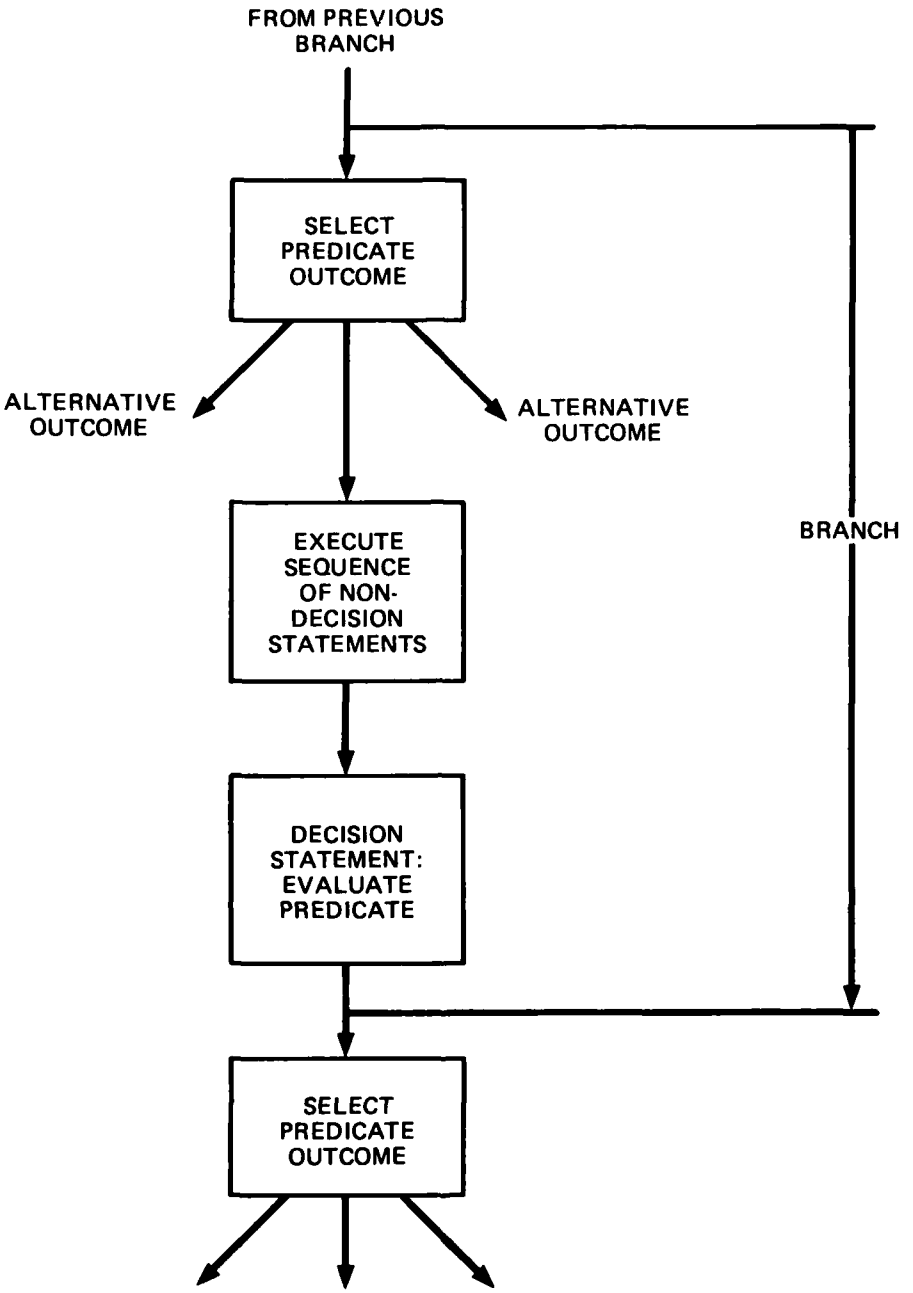



Figure 1-1. Diagram of a Branch(1)

1. C. Gannon, JOVIAL J73 Automated Verification System - Study Phase, prepared by General Research Corporation, Rome Air Development Center, TR-80-261, August 1980.

Decision-to-Decision Path (DD-path) -

See definition for "Branch".

Directed Graph -

A special type of mathematical graph characterized by each branch having a specified direction between its connecting nodes and having one or more identified entry points and one or more identified exit points.

Dynamic Analysis -

A debugging or testing technique used to evaluate a program based on its execution. The program is executed with data, the program output, and any additional execution-time reports, and is analyzed for conformity to functional or structural performance specifications.

Global Set/Use Matrix -

An NxM matrix which identifies global variables used and set within each module of a software program (where N represents the number of global variables and M represents the number of modules in the program).

Loop -

A unique sequence of one or more nodes defined on a graph in which the terminal node of the sequence is equal to the initial node of the sequence.

Mathematical Graph -

A collection of points (or nodes) $x_1$, $x_2$, ..., $x_n$ denoted by the set X, and a collection of arcs $a_1$, $a_2$, ..., $a_n$ denoted by the set A joining some or all of the nodes. Therefore, a graph is fully described and denoted by (X,A).

Module -

A logically self-contained and discrete part of a computer program. In JOVIAL J73, a module is the smallest entity that can be separately compiled.

Path -

A unique sequence of one or more program elements defined on a graph beginning with an entry point and ending with an exit point. A continuous sequence of control flow (branches) between two points in a program (usually between a program unit's entry and exit).

Program -

A collection of statements that can be assembled or compiled and can be executed as a single entity.

Program Element -

A unit within the hierarchy of software components, such as a routine, procedure, DD-path (branch), module, or statement. A DD-path is the program element most commonly used in this document.

Reachability Matrix -

An NxN matrix where N equals the number of program elements and where any (i,j) position is represented on the matrix by a 1 if program element j can be reached either directly or indirectly from program element i. If program element j cannot be reached either directly or indirectly from program element i, then position (i,j) is represented by a 0.

Retest -

The act of rerunning certain tests to verify that a change in one area of the existing software does not create data and or logic conditions that could affect the proper execution of another area.

Segments -

A contiguous sequence of executable statements with one entry point at the beginning and one exit point at the end. By executing the first statement in any segment, all other statements in that segment are also executed.

Set -

A term used to describe a data dependency that occurs when a value is placed into a variable's storage location. An example of a statement where the variable X is "set" is: X=5.

Set and Use -

A term used to describe a data dependency in which a variable is both set and used in the same statement. An example of a statement in which the variable X is both "set and used" is: X=X+1.

Set/Use Table -

A table used to analyze the flow of data (data dependency) within a program. This table has size NxM where N equals the number of variables (data elements) and M equals the number of executable statements in the module. Data within the table represents whether or not a particular variable is used, set, or both set and used indicated by a "U", "S", or "X", respectively.

Statement -

A unit of a computer program consisting of a meaningful arrangement of basic language elements which expresses a unified instruction or information, analogons to a sentence in English (1).

---

1. Gloss-Soler, S.A., The DACS Glossary - A Bibliography of Software Engineering Terms, Rome Air Development Center, Data Analysis Center for Software, GLOSS-1, October 1979.

Static Analysis -

A program analysis technique which does not actually execute the program using input data. The technique is usually employed to detect inconsistencies in semantics or in asserted versus actual conditions.

Testbed -

The set of testcases developed or modified during the various levels of testing to validate a computer program or computer program component.

Test Case Cross Reference Matrix -

A matrix used to identify the DD-paths of a program that are exercised by any given test case. It is a boolean matrix with a 1 in any (i,j) position if DD-path i is executed by test case j; otherwise it is represented by a 0.

Transitive Closure -

A mathematical technique used to derive the reachability matrix from the connectivity matrix by finding the sum of a sufficiently large number of powers of the connectivity matrix where all additions and multiplications are Boolean.

Use -

A term used to describe a data dependency that occurs when the storage location of a variable is accessed, the contents read, and the value used in comparing or computing some other value. An example of a FORTRAN executable statement where the variable Y is "used" is: X=Y*2.

# APPENDIX G - ACRONYMS

| | |
|---|---|
| CSC | Computer Sciences Corporation |
| DoD | Department of Defense |
| IMPAIR | Impossible Transfer Pairs Detection Capability |
| RADC | Rome Air Development Center |
| SREM | Software Requirements Engineering Methodology |
| SRS | Software Retest System |

# *MISSION*
# *of*
# *Rome Air Development Center*

*RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence ($C^3I$) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.*


2-8
DT